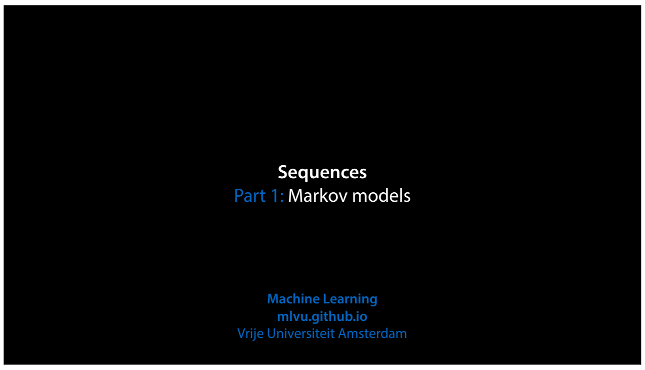

Sequences
Part 1: Markov models

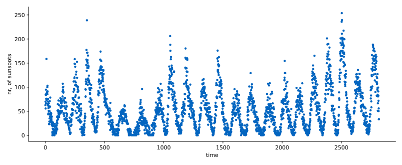
Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam

In this lecture we'll look at data that naturally forms a sequence. Language, music, stock prices. All of these can be modelled most naturally as a sequence of tokens of information coming in one after the other.

Before we look at how to model sequences, we'll look at some basic things to take into account when interpreting such data.

---

numeric 1-dimensional



2

We'll start by looking at the different types of sequential datasets we might encounter.

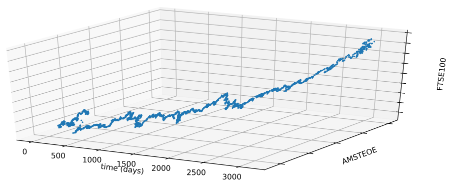
As with the traditional setting (a table of independently sampled instances) we can divide our features into numeric and discrete.

A single 1D sequence might look like this. We could this of a stock price over time, traffic to a webserver, or atmospheric pressure over Amsterdam.

In this case, the data shows the number of sunspots observed over time.

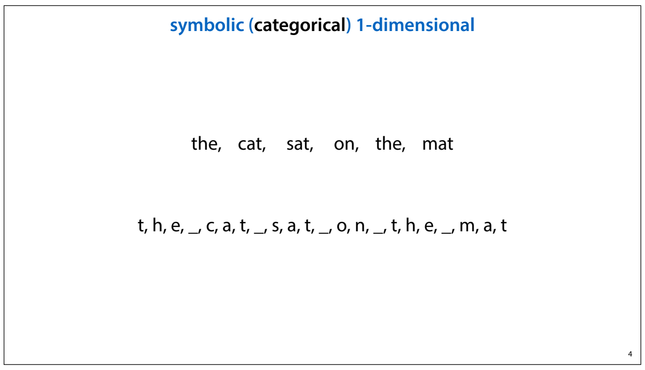
---

numeric n-dimensional



3

Sequential numeric data can also be multidimensional. In this case, we see the closing index of the AEX and the FTSE100 over time. This data is a sequence of 2D vectors.

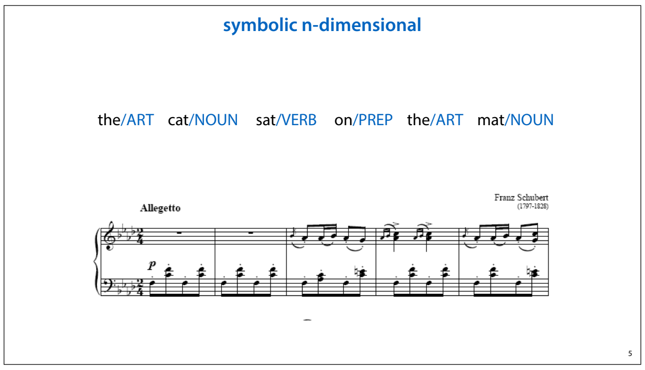

symbolic (categorical) 1-dimensional

the, cat, sat, on, the, mat

t, h, e, _, c, a, t, _, s, a, t, _, o, n, _, t, h, e, _, m, a, t

4

If the elements of our data are discrete (analogous to a categorical feature), it becomes a sequence of symbols. Language is a prime example. In fact, we can model language as a sequence in two different ways: as a sequence of **words**, or as a sequence of **characters**.



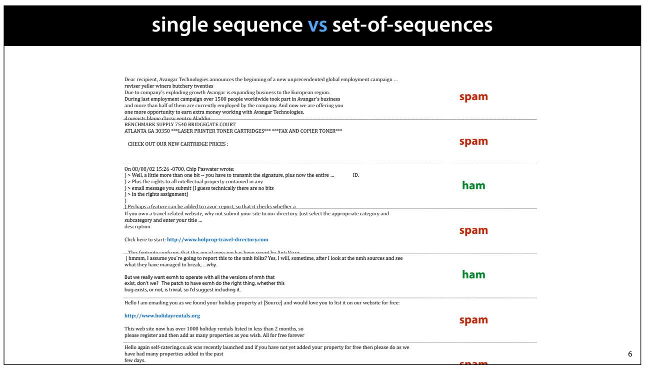
symbolic n-dimensional

the/ART cat/NOUN sat/VERB on/PREP the/ART mat/NOUN

Allegetto
Franz Schubert
(1797-1828)
p

We can also encounter sequences with multiple categorical features per timestamp.

For instance. music, or tagged language. The more complex the sequence grows, the more difficult it can be to represent. We'll stick with simple examples for this lecture.



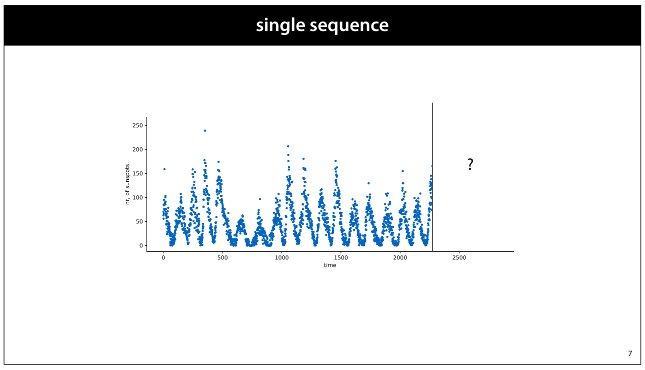
single sequence vs set-of-sequences

Dear recipient, Avangar Technologies announces the beginning of a new unprecendented global employment campaign ... reviser yeller winers butchery twenties Due to company's exploding growth Avangar is expanding business to the European region. During last employment campaign over 1500 people worldwide took part in Avangar's business and more than half of them are currently employed by the company. And now we are offering you one more opportunity to earn extra money working with Avangar Technologies. druggists blame classy gentry Aladdin	spam
BENCHMARK SUPPLY 7540 BRIDGEGATE COURT ATLANTA GA 30350 ***LASER PRINTER TONER CARTRIDGES*** ***FAX AND COPIER TONER*** CHECK OUT OUR NEW CARTRIDGE PRICES :	spam
On 08/08/02 15:26 -0700, Chip Paswater wrote:) > Well, a little more than one bit -- you have to transmit the signature, plus now the entire ... ID.) > Plus the rights to all intellectual property contained in any) > email message you submit (I guess technically there are no bits) > in the rights assignment))) Perhaps a feature can be added to razor-report, so that it checks whether a	ham
If you own a travel related website, why not submit your site to our directory. Just select the appropriate category and subcategory and enter your title ... description. Click here to start: http://www.holprop-travel-directory.com This footnote confirms that this email message has been swept by Anti Virus	spam
| hmmm, I assume you're going to report this to the nmh folks? Yes, I will, sometime, after I look at the nmh sources and see what they have managed to break, ...why. But we really want exmh to operate with all the versions of nmh that exist, don't we? The patch to have exmh do the right thing, whether this bug exists, or not, is trivial, so I'd suggest including it.	ham
Hello I am emailing you as we found your holiday property at [Source] and would love you to list it on our website for free: http://www.holidayrentals.org This web site now has over 1000 holiday rentals listed in less than 2 months, so please register and then add as many properties as you wish. All for free forever	spam
Hello again self-catering.co.uk was recently launched and if you have not yet added your property for free then please do as we have had many properties added in the past few days.	spam

Then there is the question of what we're trying to predict.

One possibility is that we have a normal classification or regression task, but the instances are not represented by feature vectors bu by sequences. Tis slide shows a simple example: email classification. Each email is a sequence (of words or or characters), and each carries one target label (ham or spam).

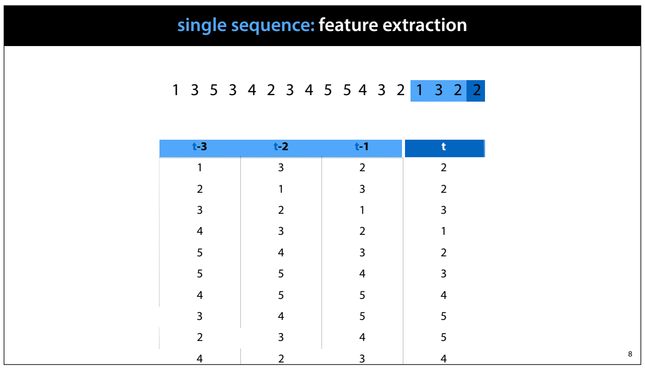
Among the instances themselves, there is not any strong sequential ordering. Emails do have a timestamp, but this ordering is usually ignored.



single sequence

An entirely different setting is one where the dataset as a whole is a sequence, and the instances are the elements in the sequence. For instance, in our sunspot example, we may consider each point in our sequence as a single instance consisting of a single feature.

In that case, we often want to predict the future values of the sequence based on what we've seen in the past.



single sequence: feature extraction

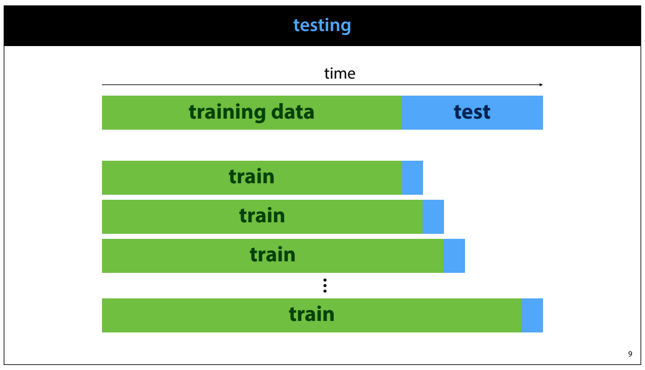
1 3 5 3 4 2 3 4 5 5 4 3 2 1 3 2 2

t-3	t-2	t-1	t
1	3	2	2
2	1	3	2
3	2	1	3
4	3	2	1
5	4	3	2
5	5	4	3
4	5	5	4
3	4	5	5
2	3	4	5
4	2	3	4

One simple way to achieve this, is to translate it to a classic regression problem by representing each point by a fixed number of values before it; in this case the 3 preceding values.

This gives us a table with a target label (the value at time t) and 3 features (the 3 preceding values). With this data in hand we can grab any standard regression model, train it, and use it on the values we're currently observing, to give us a prediction for the future.

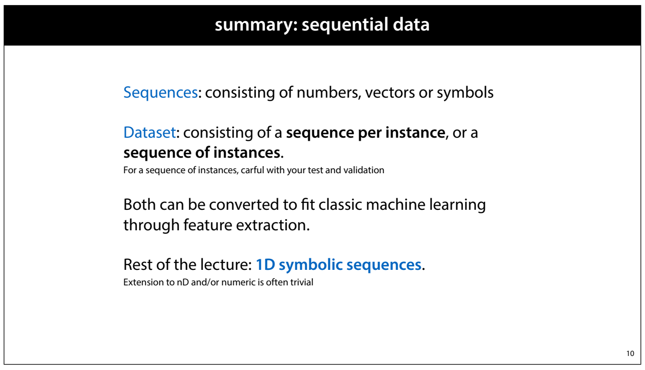
Many other features are possible: the mean over the whole history. The mean over the last 10 points, the variance over the last 10 points, and so on. This is a great way to solve this kind of sequence prediction task by translating it to a known abstract task, rather than designing a whole new approach, specific to the sequence setting.



testing

However, remember what we said in lecture 3: when your data has a meaningful ordering in time, you should keep it ordering in that way when you make data splits. You don't want to train on data that is in the future comopared to your test data. In production, you won't have that luxury, so to make your test setting a good simulation of production, you should keep your data ordered by time.

If you can expect to retrain you model periodically, then you can simulate this in your test split, by retraining after every batch of instances instances you've seen of the test set, and adding them to the training data. This is called walk forward validation.



summary: sequential data

Sequences: consisting of numbers, vectors or symbols

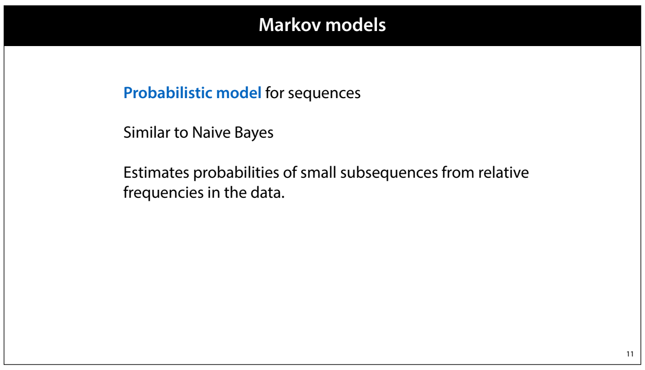
Dataset: consisting of a **sequence per instance**, or a **sequence of instances**.
For a sequence of instances, carful with your test and validation

Both can be converted to fit classic machine learning through feature extraction.

Rest of the lecture: **1D symbolic sequences**.
Extension to nD and/or numeric is often trivial

10

In the rest of the lecture, we'll cover with methods that deal with sequences natively, without the need for feature extraction.



Markov models

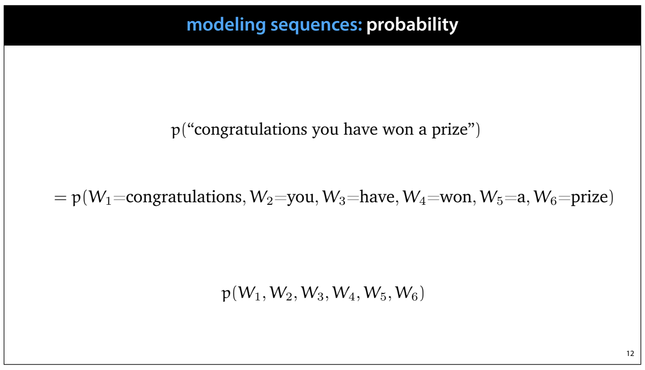
Probabilistic model for sequences

Similar to Naive Bayes

Estimates probabilities of small subsequences from relative frequencies in the data.

11

The first method we will look at is Markov modelling.



modeling sequences: probability

$$p(\text{“congratulations you have won a prize”})$$

$$= p(W_1{=}\text{congratulations}, W_2{=}\text{you}, W_3{=}\text{have}, W_4{=}\text{won}, W_5{=}\text{a}, W_6{=}\text{prize})$$

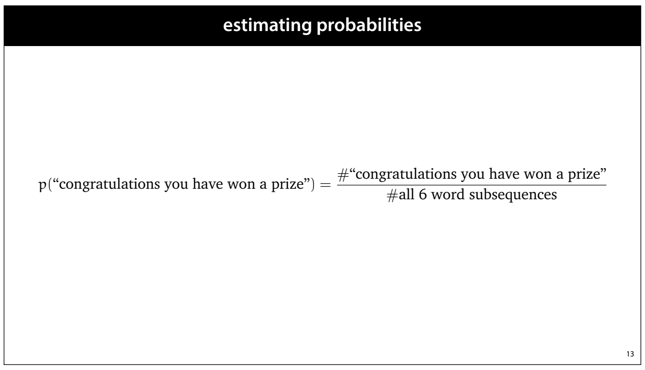
$$p(W_1, W_2, W_3, W_4, W_5, W_6)$$

12

The fundamental idea, here, is that we want to model the probability of a sequence occurring.

When modelling probability, we usually break the sequence up into its **tokens** (in this case the words of the sentence) and model each as a random variable. Note that these random variables are decidedly *not* independent: if the previous word is an article like “a”, you’re much more likely to see a noun like “prize” following it, than another article.

This leaves us with a joint distribution over 6 variables, which we would somehow like to model and fit to a dataset. How do we use our dataset to estimate the probability that we’ll see this sentence in the future?



estimating probabilities

$$p(\text{“congratulations you have won a prize”}) = \frac{\#\text{“congratulations you have won a prize”}}{\#\text{all 6 word subsequences}}$$

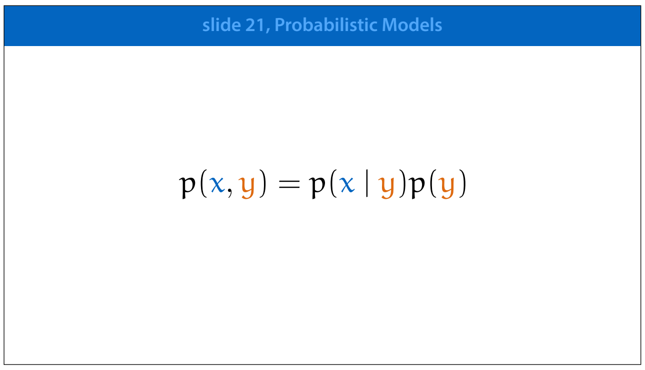
13

One simple trick we’ve used in the past to estimate probabilities is to take the relative frequencies of occurrences in the data.

We could collect a large amount of natural language data and simply count how often the sequence “congratulations you have won a prize” occurs in the data, and then divide it by the total number of 6 word sequences in the data.

The problem is that we’d need an extremely large amount of data for all sequences of interest to have been seen, and if our sequences get longer, like full emails, we’ll have no chance of collecting a dataset where every email we’re interested in has been seen before.

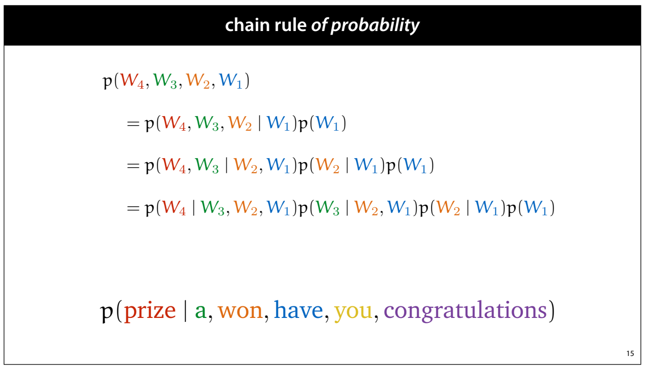
What we need to do, is break our sequence up into subsequences, estimate their probability and combine the probabilities of the subsequences, to give us the probabilty of the whole sequence.



slide 21, Probabilistic Models

$$p(x, y) = p(x \mid y)p(y)$$

To do so, we’ll use this rule from the Probabilistic Models lecture. If we have a joint distribution, we can break it up into two factors: the marginal distribution on one of the variables, times the distribution with that variable in the conditional.



chain rule *of probability*

$$\begin{aligned}
&\mathrm{p}(W_4, W_3, W_2, W_1)\\
&\quad = \mathrm{p}(W_4, W_3, W_2 \mid W_1)\mathrm{p}(W_1)\\
&\quad = \mathrm{p}(W_4, W_3 \mid W_2, W_1)\mathrm{p}(W_2 \mid W_1)\mathrm{p}(W_1)\\
&\quad = \mathrm{p}(W_4 \mid W_3, W_2, W_1)\mathrm{p}(W_3 \mid W_2, W_1)\mathrm{p}(W_2 \mid W_1)\mathrm{p}(W_1)
\end{aligned}$$

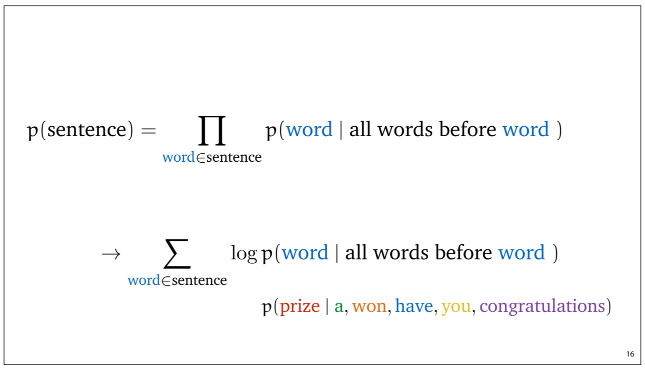
$$\mathrm{p}(\text{prize} \mid \text{a}, \text{won}, \text{have}, \text{you}, \text{congratulations})$$

15

This gives us the **chain rule of probability** (which has nothing to do with the the chain rule of calculus).

The chain rule allows us to break a joint distribution on many variables into a product of conditional distributions. In sequences, we often apply it so that each word becomes conditioned on the words before it. We could apply it in any order we like but it makes most sense to condition each word on its preceding tokens.

This tells us that if we build a model that can estimate the probability p(x|y, z) of a word x based on the words y, z that precede it, we can then *chain* this estimator to give us the joint probability of the whole sentence x, y,



$$\mathrm{p}(\text{sentence}) = \prod_{\text{word}\in\text{sentence}} \mathrm{p}(\text{word} \mid \text{all words before word})$$

$$\rightarrow \sum_{\text{word}\in\text{sentence}} \log \mathrm{p}(\text{word} \mid \text{all words before word})$$

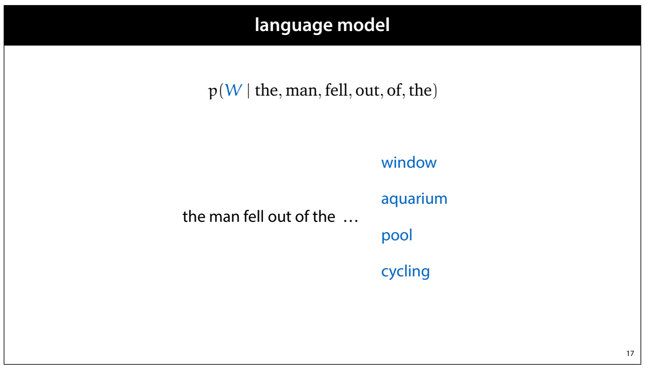
$$\mathrm{p}(\text{prize} \mid \text{a}, \text{won}, \text{have}, \text{you}, \text{congratulations})$$

16

In other words, we can rewrite the probability of a sentences as the product of the probability of each word, conditioned on its history.

If we use the log probability, this product becomes a sum. This is helpful, because these probabilities get very small, and we don't want them underflowing to zero.

This view solves part of our problem. If we can figure out how to estimate the probabilties of a particular word occurring, given all the words that precede it, we can chain these probabilities together to give us the probabilities of a whole sentence, or an even longer sequence of words (like a whole email).



language model

$$\mathrm{p}(W \mid \text{the}, \text{man}, \text{fell}, \text{out}, \text{of}, \text{the})$$

the man fell out of the …

window

aquarium

pool

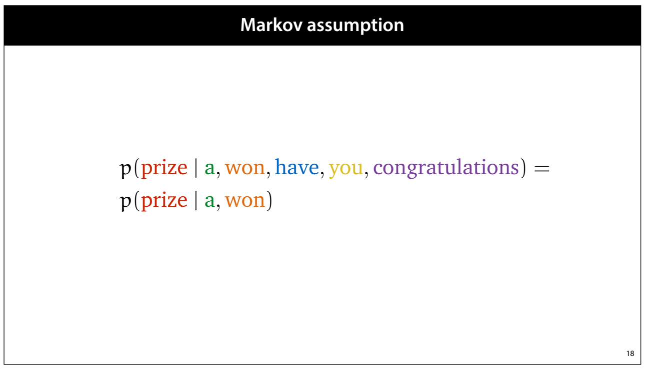
cycling

17

Note that this is no easy task.

A perfect language model would encompass everything we know about language: the grammar, the idiom and the physical reality it describes. For instance, it would give window a very high probability, since that is a very reasonable way to complete the sentence. Aquarium is less likely, but still physically possible and grammatically correct. A very clever language model might know that falling out of a pool is not physically possible (except under unusual circumstances), so that should get a lower probability, and finally cycling is ungrammatical, so that should get very low probability (perhaps even zero).

The problem is most sentences of this length will never have been seen before in their entirety. A simple way to get a basic model is **to limit how far back we look in the sentence**.



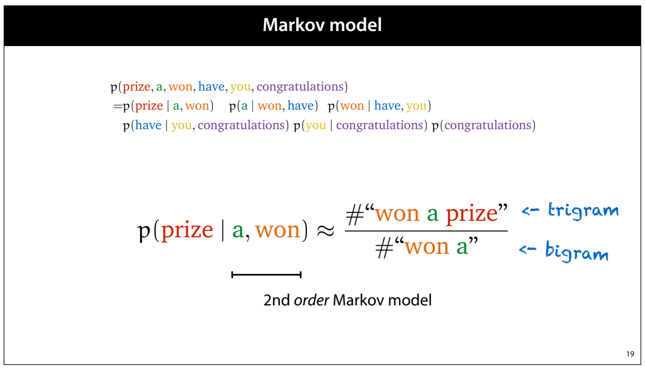
Markov assumption

$$p(\text{prize} \mid \text{a}, \text{won}, \text{have}, \text{you}, \text{congratulations}) =$$
$$p(\text{prize} \mid \text{a}, \text{won})$$

18

This is called a **Markov assumption**. We just take the probability of a word given all the words that precede it, and we assume that it's equal to the probability of the word conditioned only on the k words that precede it.

This is a bit like the naive Bayes assumption: we know it's incorrect, but it still yields a very usable model. The number of words we retain in the conditional is called the order of the Markov model: this is a Markov assumption for a second-order Markov model.



Markov model

$$\begin{aligned} &p(\text{prize}, \text{a}, \text{won}, \text{have}, \text{you}, \text{congratulations}) \\ &= p(\text{prize} \mid \text{a}, \text{won}) \;\; p(\text{a} \mid \text{won}, \text{have}) \;\; p(\text{won} \mid \text{have}, \text{you}) \\ &\quad p(\text{have} \mid \text{you}, \text{congratulations}) \; p(\text{you} \mid \text{congratulations}) \; p(\text{congratulations}) \end{aligned}$$

$$\underbrace{p(\text{prize} \mid \text{a}, \text{won})}_{\text{2nd } \textit{order} \text{ Markov model}} \approx \frac{\#\text{“won a prize”}}{\#\text{“won a”}} \quad \begin{matrix} \text{<- trigram} \\ \text{<- bigram} \end{matrix}$$

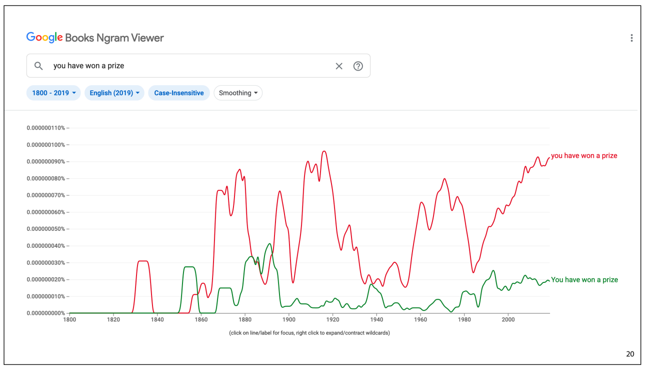
19

Using the Markov assumption and the chain rule together, we can model a sequence as limited-memory conditional probabilities. These probabilities can then be very simply estimated from a large dataset of text (called **a corpus**).

To estimate the probability of prize given "won a" we just count how often "won a prize" occurs as a proportion of the times "won a" occurs. In other words, how often "won a" is followed by prize.

These n-word snippets are called **n-grams**. "won a prize" is a **trigram**, and "won a" is a **bigram**.

This type of language model is often called a **Markov model**, because of the Markov assumption of limited memory. The size of the memory is referred to as the **order** of the Markov model. The higher the order of your model, the more you can model, but the more data you'll need, to make sure that you've seen all the n-grams you're interested in often enough.



you have won a prize

1800 - 2019 English (2019) Case-Insensitive Smoothing

(click on line/label for focus, right click to expand/contract wildcards)

20

With the kind of datasets you can download and run yourself, you can estimate good statistics for brigrams and trigrams. If you have a larger corpus, like Google's corpus of all books, you can easily go up to 5-grams.

what does this buy us?

Sequence classification
sequences as instances

Sequence prediction
tokens as instances

So, now that we have worked out a first way to approximate probabilities for sequences, what can we do with this?

We can use this to tackle both the case where our data consists of a separate sequence per instance (like in our spam classification example), and the case where our data consists of a single sequence, and we're trying to predict the next token.

sequence classification

$$p(\text{spam} \mid \text{“congratulations, you have won a prize”})$$

We'll start with a sequence-per-instance example: the spam classification task. We'll see how to approach this with a Markov model.

Ultimately, what we want to know is the the probability of the class, given the contents of the message.

Bayes classifier

$$p(\text{spam} \mid W_1, \ldots, W_n) \propto p(W_1, \ldots, W_n \mid \text{spam})p(\text{spam})$$

We'll train a generative classifier. First, we'll use Bayes rule to flip around the probabilties.

The marignal probability p(spam) we can estimate as as the proportion of spam emails in our data set. For the probability of the message given the class, we'll use our language model.

conditional on class

$$\begin{aligned}
&p(\text{prize}, \text{a}, \text{won}, \text{have}, \text{you}, \text{congratulations} \mid \text{spam}) \\
=&p(\text{prize} \mid \text{a}, \text{won}, \text{spam})\ p(\text{a} \mid \text{won}, \text{have}, \text{spam}) \\
&\ p(\text{won} \mid \text{have}, \text{you}, \text{spam})\ p(\text{have} \mid \text{you}, \text{congratulations}, \text{spam}) \\
&\ p(\text{you} \mid \text{congratulations}, \text{spam})\ p(\text{congratulations}, \text{spam})
\end{aligned}$$

$$p(\text{prize} \mid \text{a}, \text{won}, \text{spam}) \approx \frac{\#_{\text{spam}}\text{“won a prize”}}{\#_{\text{spam}}\text{“won a”}}$$

We use the chain rule and the Markov assumption to define the probability that a message occurs. This is exactly as before, except that now, everything is also conditioned on the class spam.

We then estimate the different conditional probabilities by computing the relative frequencies of bigrams and trigrams, as before, but we compute them only over the **spam** part of our data.

algorithm: 2nd-order markov model classifier

```
training:
  for each class c:
    for n in 1, 2, 3:
      count n-grams in all text belonging to class c

classification:
given text w1, …, wm
```

$$p(c \mid w_1, \ldots, w_m) \propto p(w_1, \ldots, w_m \mid c)p(c)$$

$$p(w_1, \ldots, w_m \mid c) =$$
$$p(w_1 \mid c)\, p(w_2 \mid w_1, c) \prod_{i \in [3, \ldots, m]} p(w_i \mid w_{i-1}, w_{i-2}, c)$$

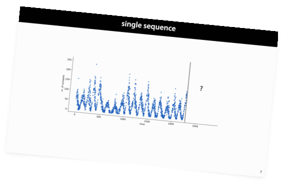
25

Here is the complete algorithm, for a classifier using a second order Markov model. First, we split our data by class. We will train a separate language model for each class.

Then, in each of these subsets, we count all occurrences of all unigrams, bigrams and trigrams. This is all the “training” we do.

Then, to classify a new sequence, we need to compute the probability of the sequence given the class, and multiply it by the class marginal probability.

In practice, as noted before, we use log probabilities, to keep low probability values from underflowing.

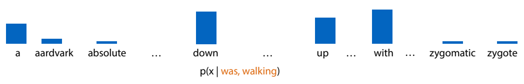


26

We can also use a Markov model on unlabelled data, to predict the future. In this case, all we need is simply a large amount of natural language text.

seed: [i, was, walking]

probability p(x | i, was, walking) = p(x | was, walking)



sample: x = down

new seed: [i, was, walking, down]

Loop

27

sequential sampling from a language model

start with a small *seed* sequence $s = [w_1, w_2, w_3]$ of tokens.

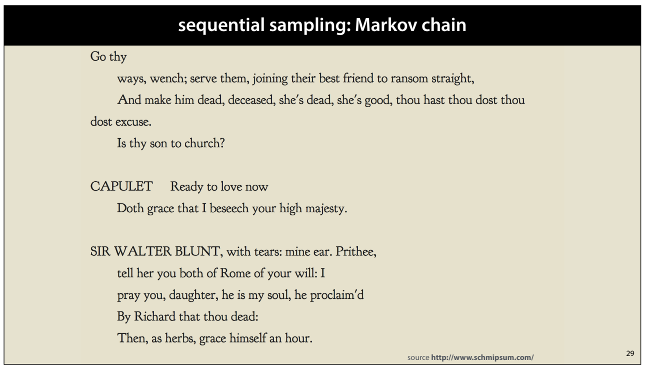
```
loop:
  Sample next word w according to p(W = w | w1, w2, …)
  append w to s
```

28

One interesting thing we can do with such a Markov model, is to sample from it, step by step. We start with a seed of a few words, and then work out the probability distribution over the next word, given the last n words. We sample from this distribution, append the sample to our text and repeat the process.

Note that with the Markov assumption, we only need the last n elements of the sequence to work out the probabilities.

Sequential sampling is also known as **autoregressive sampling**. In the context of Markov models, the sampling process is often called a **Markov chain**.



sequential sampling: Markov chain

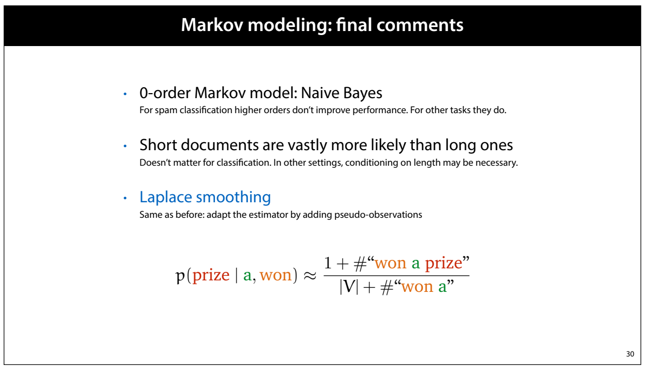
Go thy
    ways, wench; serve them, joining their best friend to ransom straight,
    And make him dead, deceased, she's dead, she's good, thou hast thou dost thou
dost excuse.
    Is thy son to church?

CAPULET   Ready to love now
    Doth grace that I beseech your high majesty.

SIR WALTER BLUNT, with tears: mine ear. Prithee,
    tell her you both of Rome of your will: I
    pray you, daughter, he is my soul, he proclaim'd
    By Richard that thou dead:
    Then, as herbs, grace himself an hour.

source **http://www.schmipsum.com/**

Here is is a bit of text sampled from a Markov model trained on the works of Shakespeare. Even with such a simple language model, we can see some quite realistic patterns appearing.



Markov modeling: final comments

- 0-order Markov model: Naive Bayes
  For spam classification higher orders don't improve performance. For other tasks they do.
- Short documents are vastly more likely than long ones
  Doesn't matter for classification. In other settings, conditioning on length may be necessary.
- Laplace smoothing
  Same as before: adapt the estimator by adding pseudo-observations

$$p(\text{prize} \mid \text{a}, \text{won}) \approx \frac{1 + \#\text{“won a prize”}}{|V| + \#\text{“won a”}}$$

V is the vocabulary (the set of all n-grams to the language model). As before, we can give the pseudo observations a smaller weight than 1, to have less impact on the estimate.

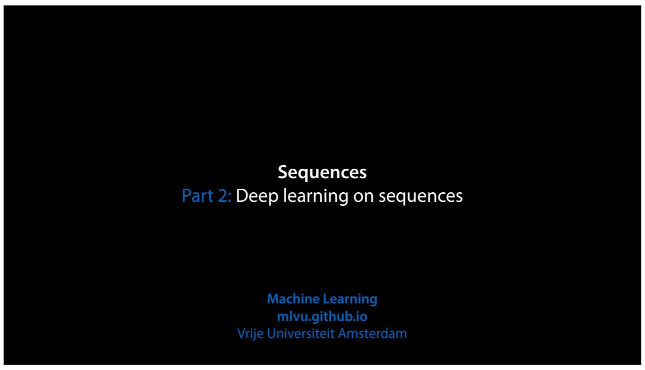
Garkov
by Josh Millard via Jim Davis
I'D SAY YOU'RE FAT!
LOOK AT THOSE PEOPLE GOING TO TAKE OVER THE
HONK! HONK!
THAT'S ALL
Garkov
by Josh Millard via Jim Davis
YEEOW!
CLUNK!
MY LEGS!
NOTIFY YOUR NEXT OF KIN

Garfield generated by a Markov model

source: **https://blog.codinghorror.com/markov-and-you/**

Sequential sampling can lead to amusing, results, but it's unlikely to fool a human reader for very long. If we apply

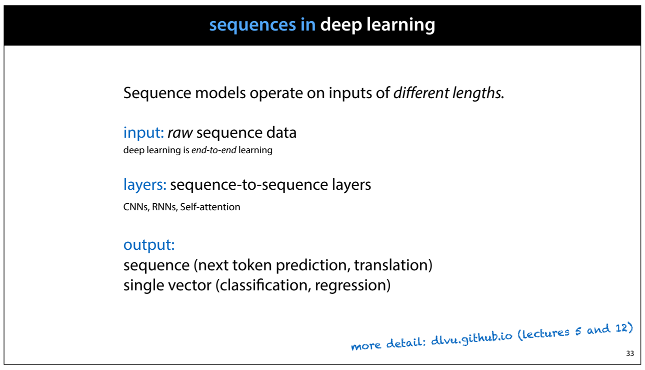
In the remainder of the lecture, we'll look at ways of dealing with sequences in a deep learning setting.



Sequences
Part 2: Deep learning on sequences

Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam

In this video we'll look at



sequences in deep learning

Sequence models operate on inputs of *different lengths.*

input: *raw* sequence data
deep learning is *end-to-end* learning

layers: sequence-to-sequence layers
CNNs, RNNs, Self-attention

output:
sequence (next token prediction, translation)
single vector (classification, regression)

more detail: dlvu.github.io (lectures 5 and 12)

33

The basic idea of sequence models is very similar to other deep learning model. The main characteristic that we need to ensure is that the model can handle input sequences of **different lengths**.

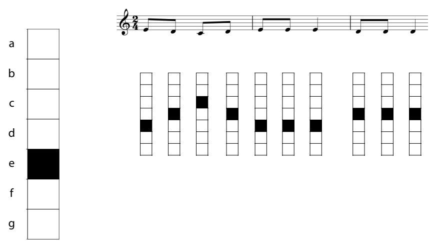
We feed the model with raw data, with no feature extraction, so that we don't lose any information. We build our model out of sequence-to-sequence layers, of which we'll see three examples in this lecture. These take a sequence of vectors as their input, and produce a sequence of vectors as their out

And finally, we need to produce some output. We can either produce a sequence of the same length as the input, for instance if we want to predict the next token at each stage, or we can output a single vector to represent the whole sequence, for instance when we want to classify the sequence.

We'll look at each of these three stages in detail.

As before, if you are actually implementing these things, you'll need some details that we won't discuss here. You can go to **dlvu.github.io** to see our lectures for the MSc course deep learning, where we discuss the same subjects, but provide some of the fines details too.

inputs

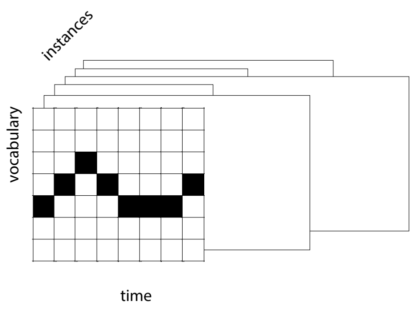


34

First, the **input**. As we've seen, when we want to do deep learning, our input should be represented as a tensor. Preferably in a way that retains all information (i.e. we want to be learning from the raw data, or something as close to it as possible).

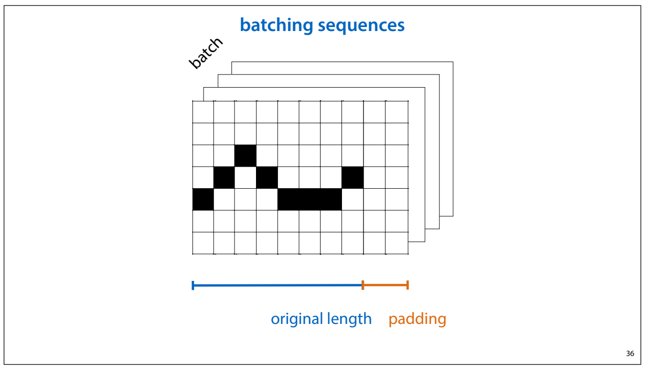
Here is an example: to encode a simple monophonic musical sequence, we just one-hot encode the notes, and encode the note sequence as a matrix: one dimension for time, one dimension for the notes. We can do the same thing for characters or even words in natural language sequences.

source: **https://violinsheetmusic.org**



35

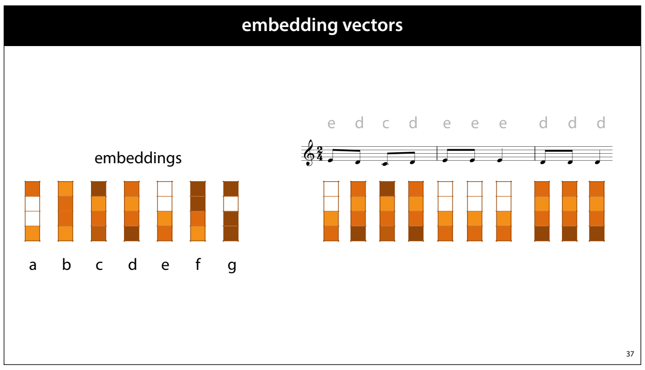
One thing is different from what we've seen so far in deep learning data: if we have multiple sequences of different lengths, this leads to a data set of matrices of different sizes.



In principle, this is not a problem, we want to build models that can deal with sequences of any length (and can generalise over sequences of variable lengths), so they should be able to handle this.

However, within a batch it's usually required that all sequences have the same length. One way to deal with this, is to sequences of similar length together (for instance by sorting the data by length) and then pad the shorter sequences with zero vectors, so that all sequences are the same length.

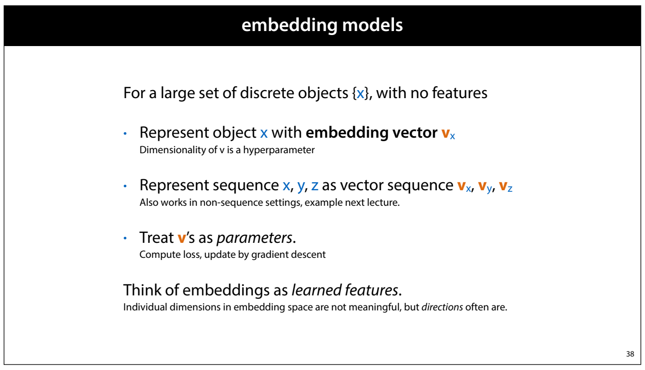
At this point, we have translated a batch of input sequences into a 3-tensor, which can be consumed by any deep learning model.



One-hot vectors are fine if if you have a small vocabulary of symbols (like seven notes), but if you want to model 100 000 words, you're using a lot of memory that is mostly filled with zeros.

An alternative method is to use **embedding vectors**. The idea here is that you assign each input symbol in your vocabulary a vector of random values. You then translate a symbolic input sequence into a sequence of vectors by mapping the input symbold to their corresponding embedding vectors. The dimensionality of the embedding vectors is a hyperparameter, but it's usually set between 64 and 1024.

The fundamental trick of embedding vectors is that we treat these vectors **as parameters of the model**. We feed this input sequence to the model (we'll describe what that looks like later), compute the loss, backpropagate, and we get gradients on all parameters of the model, including these embedding vectors. As we train, these vectors become useful representations of our words in some high dimensional space.



embedding models

For a large set of discrete objects {x}, with no features

- Represent object x with **embedding vector** v_x
  Dimensionality of v is a hyperparameter
- Represent sequence x, y, z as vector sequence v_x, v_y, v_z
  Also works in non-sequence settings, example next lecture.
- Treat v's as *parameters*.
  Compute loss, update by gradient descent

Think of embeddings as *learned features*.
Individual dimensions in embedding space are not meaningful, but *directions* often are.

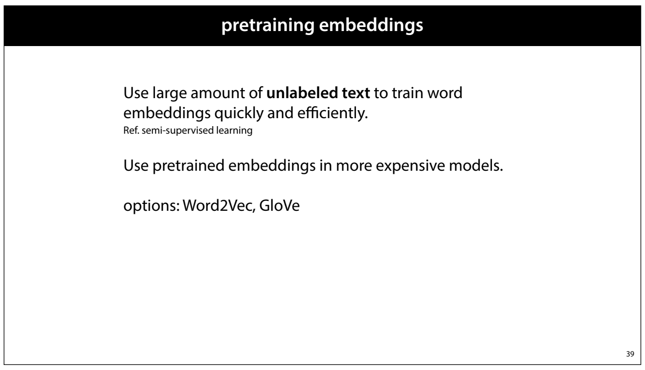
38

Embedding vectors occur in many contexts, so let's define them more broadly. In any setting where you have a large collection of discrete objects, and no features for these objects, you can represent them with embedding vectors. You assign a unique vector to each object in your set, and use these vectors to represent the objects you want to learn over. If you are training on a sequence of objects, you turn this into a sequence of embedding vectors.

Once you've computed your loss, you update the values of the embedding vectors by gradient descent, possibly using backpropagation to compute the gradients.

We can think of the embeddings as learned features:

we don't have features for our objects, so we simply assign them some random features, and then tweak the values of these by gradient descent.



pretraining embeddings

Use large amount of **unlabeled text** to train word embeddings quickly and efficiently.
Ref. semi-supervised learning

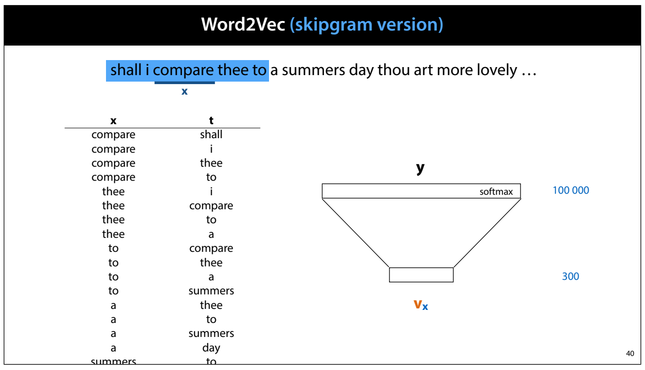
Use pretrained embeddings in more expensive models.

options: Word2Vec, GloVe

39

In addition to training embeddings together with the other parameters of our model, embeddings also provide a good opportunity for **pre-training**. If we have a large amount of unlabeled text available, and we can think of a cheap way to use it to train word embeddings, these can then be re-used in larger, more elaborate models.

We'll take a quick look at the Word2vec model as an example.



Word2Vec (skipgram version)

shall i compare thee to a summers day thou art more lovely …

x	t
compare	shall
compare	i
compare	thee
compare	to
thee	i
thee	compare
thee	to
thee	a
to	compare
to	thee
to	a
to	summers
a	thee
a	to
a	summers
a	day
summers	to

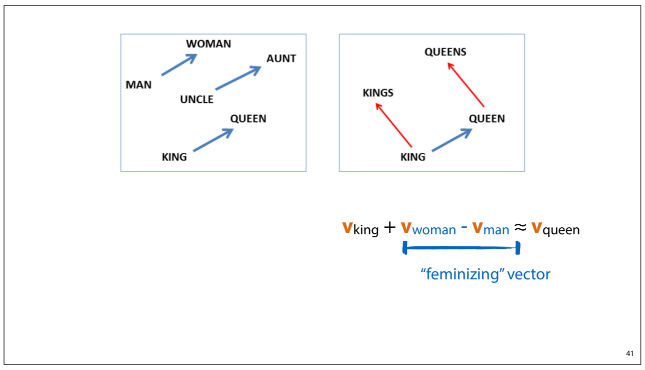
40

We start with a large corpus: a data set of natural language. From this we create a very simple dataset. We slide a window of, say, five words, along the text, and match every word to the words that appear in its context. The task is to predict, for a given input word, the distribution on the words appearing in the context.

We model this very simply by creating embedding vectors for all words in our vocabulary. We then feed these to a single-layer neural network with one output for every word in our vocabulary and with a softmax output. If we have 100 000 words in our vocabulary, then we can think of this as a one-layer classification network with 100 000 classes, and the embedding as input features. We train both the embedding and the weight of the network in concert.

If training is succesful, we know that our embedding vectors now contain the information about which words are likely to appear in their context. The basic idea is that this information captures a lot of their meaning (this is sometimes called the distributional hypothesis).

The softmax activation over 100K outputs is very expensive to compute, and you need some clever tricks to make this feasible (called *hierarchical* softmax or negative sampling). We won't go into them here.

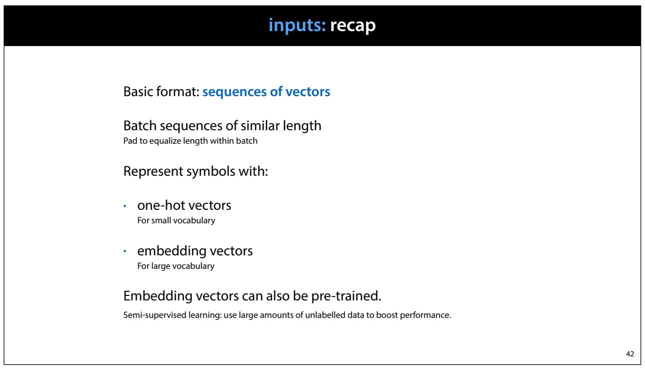


$\mathbf{v}_{\text{king}} + \underbrace{\mathbf{v}_{\text{woman}} - \mathbf{v}_{\text{man}}}_{\text{“feminizing” vector}} \approx \mathbf{v}_{\text{queen}}$

41

If we investigate what the Word2Vec embeddings look like after training, we can tease out some interesting properties. For instance, it seems like there is a direction in the resulting embedding space, that, if we move along this direction, pushes male words towards their female counterparts. What's more, if we subtract the embedding of the word woman from the embedding of the word man, we we find roughly this direction.

Compare this with the "smiling vector" we saw in the autoencoder lecture. Word2Vec isn't an autoencoder, but we are learning a similar kind of latent space model.



inputs: recap

Basic format: sequences of vectors

Batch sequences of similar length
Pad to equalize lengths within batch

Represent symbols with:

- one-hot vectors
  For small vocabulary
- embedding vectors
  For large vocabulary

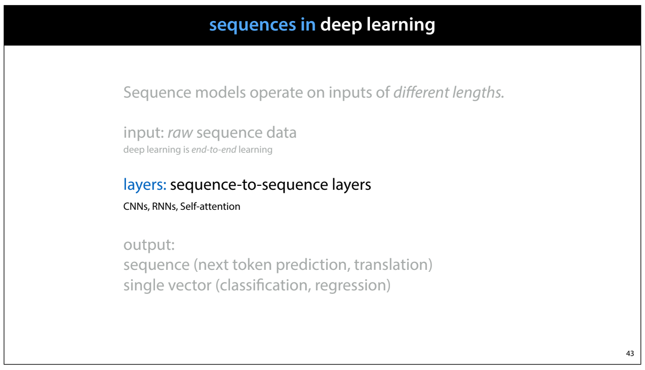
Embedding vectors can also be pre-trained.
Semi-supervised learning: use large amounts of unlabelled data to boost performance.

42

So, these are the most important methods for representing input sequences so that deep learning models can understand them. We need to somehow translate our input to a sequence of vectors.

If we have a symbolic input, we can do this by representing the symbols with **one-hot vectors** if the vocabulary is small, and with **embedding vectors** if the vocabulary is larger.

Pre-training your embedding



sequences in deep learning

Sequence models operate on inputs of *different lengths.*

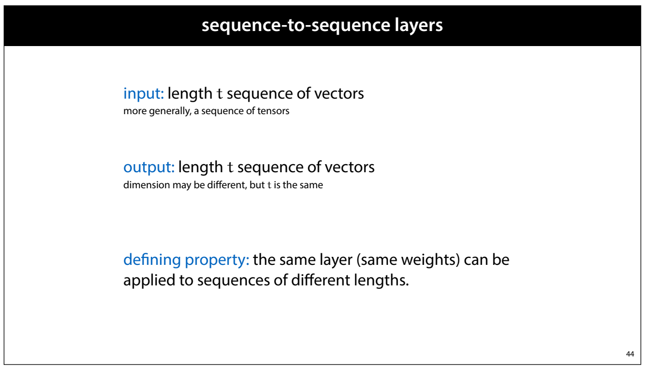
input: *raw* sequence data
deep learning is *end-to-end* learning

layers: sequence-to-sequence layers
CNNs, RNNs, Self-attention

output:
sequence (next token prediction, translation)
single vector (classification, regression)

43

Now that we know how to represent our input, we need some layers that we can stack together to build a deep neural net for sequences.



sequence-to-sequence layers

input: length t sequence of vectors
more generally, a sequence of tensors

output: length t sequence of vectors
dimension may be different, but t is the same

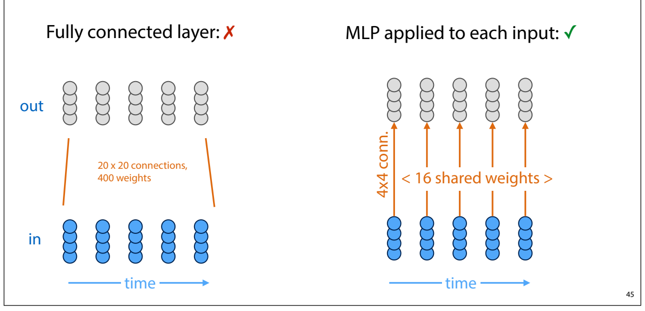
defining property: the same layer (same weights) can be applied to sequences of different lengths.

44

The next ingredient we need is layers that we can stack on top of each other. These need to be **sequence-to-sequence layers**. These are layers that take sequence of vectors as input and produce a new sequence of vectors as output. The input and output

The defining property of a sequence-to-sequence layer is that they can consume sequences of different lengths with the same set of weights. That is, in one iteration of gradient descent, we can feed the layer a sequence of 5 words, get a loss and update its weights, and the next iteration we can feed it a sequence of 15 words, and get a loss and a gradient *on the same weights*.

fully connected vs repeated MLP



Here is an example: imagine that we need a layer that consumes a sequence of five vectors with four elements each and produces another sequence of five vectors with four elements each.

A fully connected layer would simply connect every input with every output, giving us 400 connections with a weight each. This is *not* a sequence-to-sequence layer. Why not? Imagine that the next instance has 6 vectors: we wouldn't be able to feed it to this layer without adding extra weights.

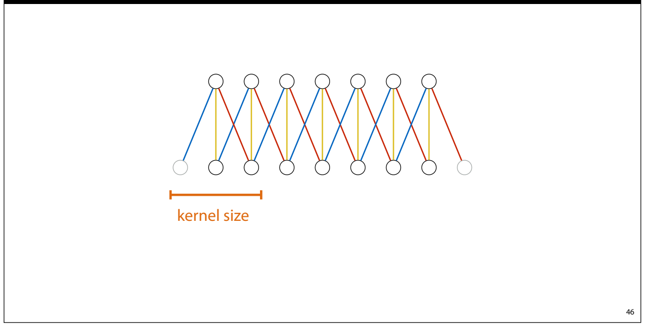
The version on the right also uses an MLP, but only applies it to each vector in isolation: this gives us 4x4=16 connections per vector and 80 in total. These eighty connection share only 16 unique weights, which are repeated at each step.

This is a sequence-to-sequence layer. If the next instance has 6 vectors, we can simple repeat the same MLP again, we don't need any extra weights. This is the basic idea of the sequence-to-sequence layer. If we see an input with a new length, we can take the layer, keep the weights the same, but configure the layer to accept a sequence of the required length.

Of course, this second option may technically be a sequence-to-sequence layer, but it doesn't actually learn over the time dimension. The value of vector 5 is in no way influenced by the values of input vectors 1 through 4, because there are no connections between them. Luckily, there is a layer that we've seen already, that is a sequence-to-sequence layer and allows for information to be propagated along the time dimension.

NB: We call the sequence dimension "time", but it doesn't necessarily always represent time.

1D Convolution



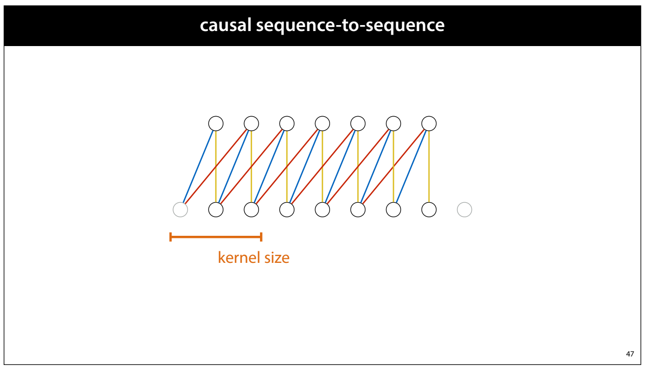
This is the 1D convolution that we first saw in the deep learning lecture.

Note that the number of (distinct) weights depends only on the size of the kernel and the number of input and output channels. If we see a longer or shorter sequence, we just repeat the same kernel more often, but we don't need extra weights.

All we need to do to fit our definition of a sequence to sequence layer is to add a little padding so that the input and output have the same length.

Note that even though we are now allowing information to propagate from one position in the input to another position in the output, we're only allowing this over a finite distance. This is a bit like the finite memory of the Markov model. We can use the history (and future) of the sequence but only a fixed, finite part of it.

NB: Note that the MLP example from the last slide is equal to a 1d convolution with a kernel size of 1.

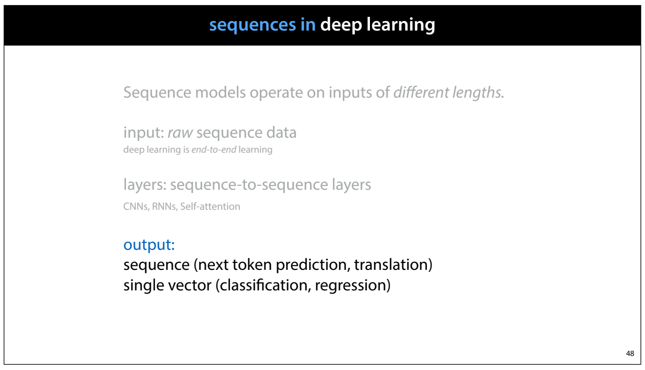


causal sequence-to-sequence

In many settings, it's not reasonable to let the model look into the future. For instance when you only have this information for your training data, but you don't expect to have it in production. In that case it's important to wire up your sequence to sequence layer so that each output node only has connections to the corresponding input node and to ones before it.

This is called **causal sequence to sequence layer**. And pictured here is a a **causal convolution**.

Note that this doesn't imply that we're performing causal inference: that is, we're not making guaranteed distinctions between correlation and causation, as we discussed in the social impact videos. It's simply a way



sequences in deep learning

Sequence models operate on inputs of *different lengths.*

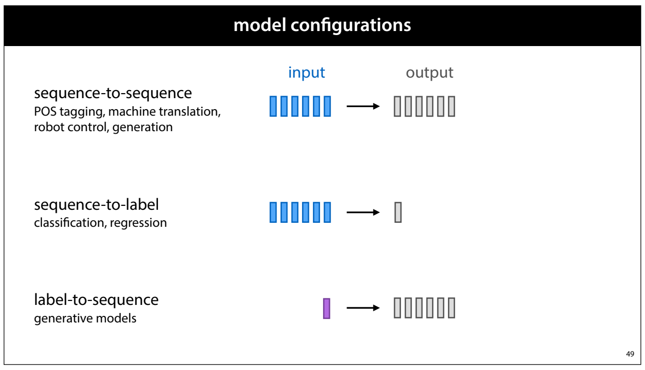
input: *raw* sequence data
deep learning is *end-to-end* learning

layers: sequence-to-sequence layers
CNNs, RNNs, Self-attention

output:
sequence (next token prediction, translation)
single vector (classification, regression)

The convolution layer is hopefully enough to give you a concrete idea of what a sequence to sequence layer looks like. In the next video we'll see another way of building seuqence to sequence layers.

So, now we have an input format: a sequence of vectors, and a type of layer, which translates a sequence of vectors to another sequence of vectors. Finally, we need to give our network some output: something that allows it to compute a single loss, so that we can start our backpropagation.



model configurations

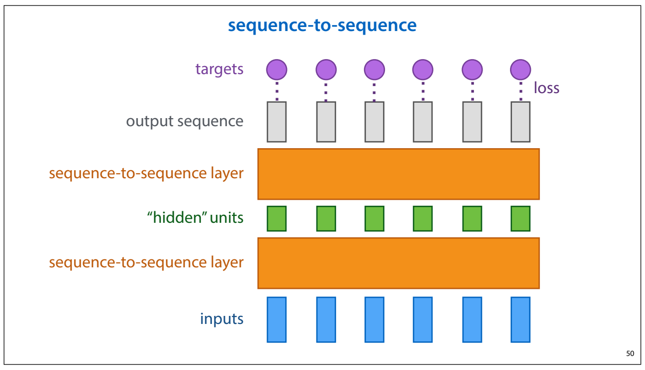
sequence-to-sequence
POS tagging, machine translation, robot control, generation

sequence-to-label
classification, regression

label-to-sequence
generative models

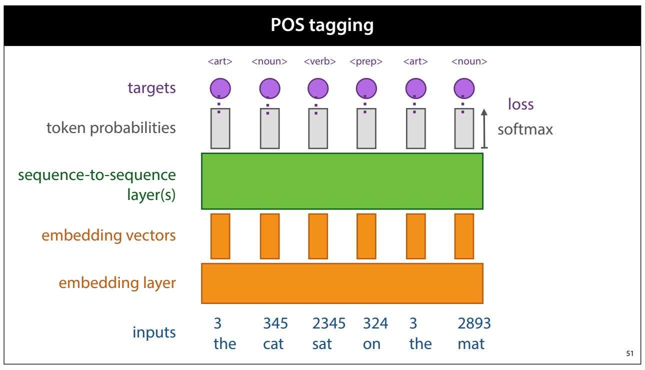
There's number of ways we can configure our model, depending on what we're trying to achieve.

Here are three basic configurations we may want to build.



A sequence-to-sequence task is probably the simplest set-up. Our dataset consists of a set of input and target sequences.

We simply create a model by stacking a bunch of sequence to sequence layers, and our loss is the difference between the target sequence and the output sequence.

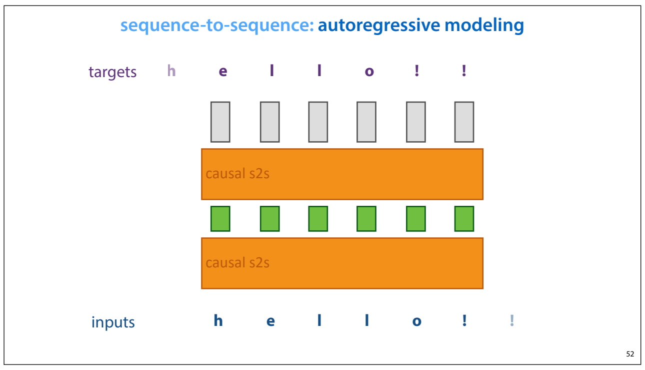


Here’s a simple example of a sequence to sequence task: tag each word in a sentence with its grammatical category. This is known as part-of-speech tagging. All we need is a large collection of sentences that have been tagged.

For the embedding layer, we convert our input sequence to positive integers. We have to decide beforehand what the size of our vocabulary is. If we keep a vocabulary of 10 000 tokens, the embedding layer will create 10 000 embedding vectors for us.

It then takes a sequence of positive integers and translates this to a sequence of the corresponding embedding vectors. These are fed to a stack of s2s layers, which produce a sequence of vectors with al many elements as output tokens. After applying a softmax activation to each vector in this sequence, we get a sequence of probabilities over the target tokens.

In the rest of the lecture we will omit the embedding layer, assuming that some suitable input representation has been chosen.

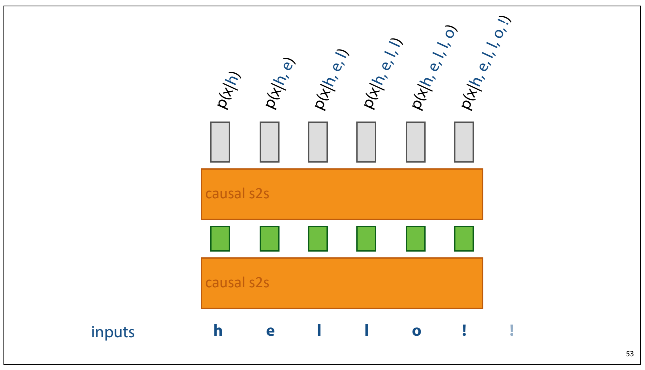


One interesting thing we can build with a sequence-to-sequence model is an **autoregressive model**.

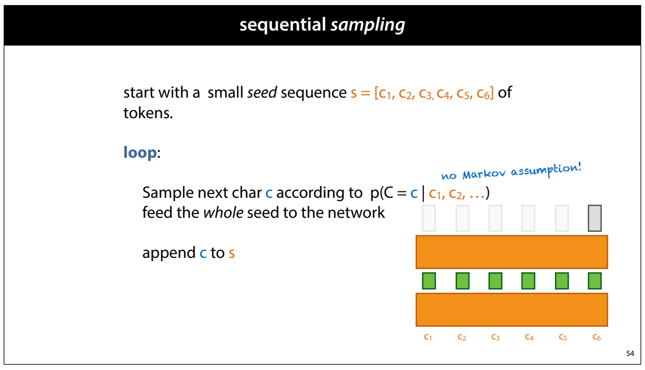
We feed some sequence, and to set the target as the same sequence, shifted one token to the left. We then feed the input through several causal layers, so that the network can only look backward in the sequence. And we produce a probability distribution on the output characters.

This effectively trains the model to predict the next character in the sequence, but it does so for the whole sequence in parallel.

Note that this only works with causal models, because non-causal models can just look ahead in the sequence to see the next character.



Each of these outputs gives us the probability of the next character in the sequence, given the preceding characters.



sequential *sampling*

start with a small *seed* sequence $s = [c_1, c_2, c_3, c_4, c_5, c_6]$ of tokens.

loop:

Sample next char c according to $p(C = c \mid c_1, c_2, \ldots)$

feed the *whole* seed to the network

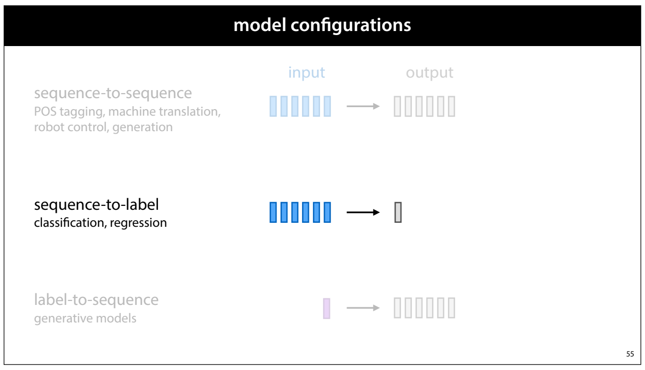
append c to s

no Markov assumption!
c1
c2
c3
c4
c5
c6
54

After the network is trained, we can start with a small seed of tokens, and sequentially sample a likely sequence. This is exactly what we did with the Markov model, but now we have a potentially much more powerful model, with a potentially infinite memory.

After training, we feed the model the whole seed each time and look only at its last output to give us a probability distribution on what the next token will be. The other outputs are only used during training.

We'll see some examples of data generated this way this after we've explained LSTM networks.



model configurations

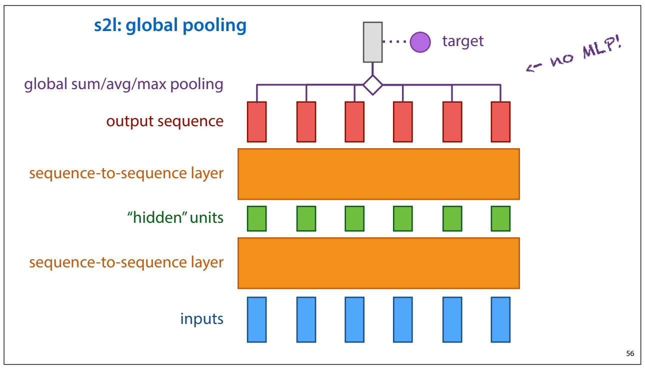
sequence-to-sequence
POS tagging, machine translation, robot control, generation

sequence-to-label
classification, regression

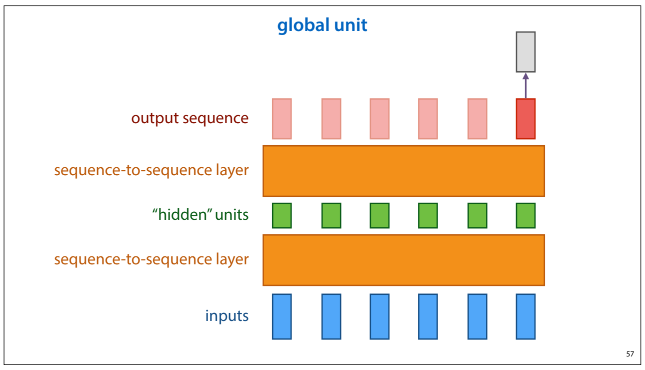
label-to-sequence
generative models

So that's what we can do with sequence-to-sequence model.

If we have a sequence labeling task, like the email spam classification example we saw earlier, We'll need to construct a model that consumes sequences of variable lengths, but at some stages reduces them down to a single label (like a class promability vector).



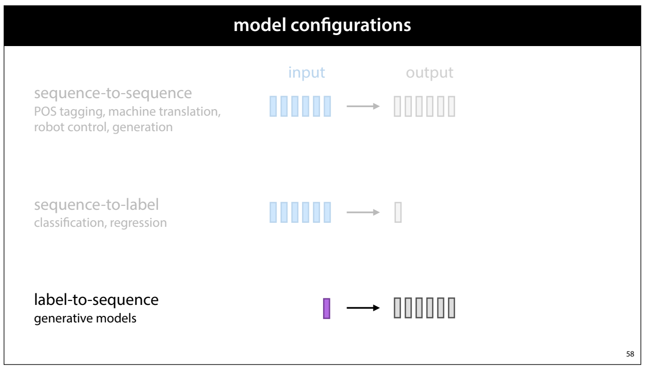
s2l: global pooling
target
<- no MLP!
global sum/avg/max pooling
output sequence
sequence-to-sequence layer
"hidden" units
sequence-to-sequence layer
inputs
56



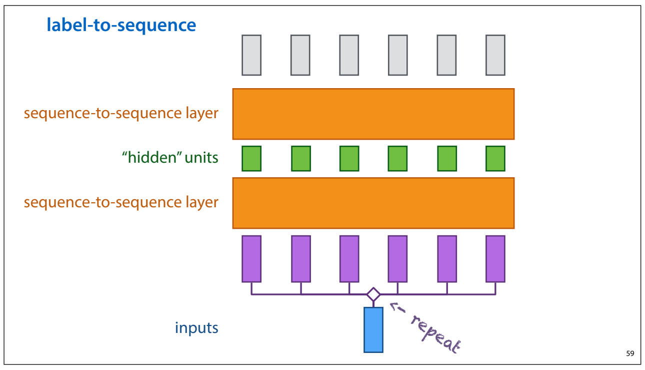
Another approach is to simply take one of the vectors in the output sequence, use that as the output vector and ignore the rest. If we train the neural network to classify only on this unit, it will hopefully learn to put the right information into this output vector.

If you have causal s2s layers, it's important that you use the last vector, since that's the only one that gets to see the whole input sequence.

For some layers (like recurrent ones), this kind of approach puts more weight on the end of the sequence, since the early nodes have to propagate through more intermediate steps in the s2s layer. For others (like self-attention), all inputs in the sequence are treated equally, and there is little difference between a global unit and a pooling layer.

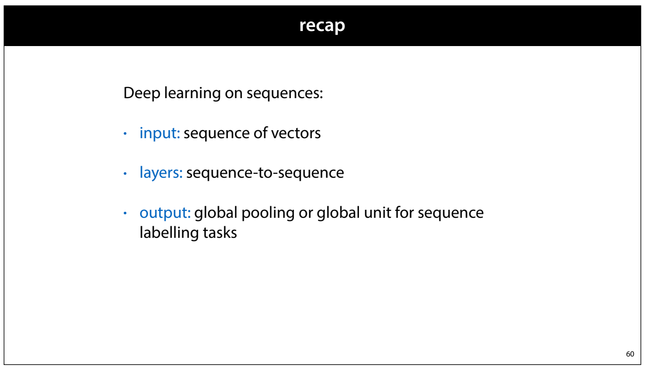


Finally, if we want to train a generative model on sequences, we may want to start with a label, for instance a latent vector, or a representation of the type of thing we want to sample, and map that to a sequence.



The simplest way to achieve this, is just to take your input vector and to repeat it into a sequence of the same vector over and over again.

For some layers, like recurrent ones, there are other ways to feed a single vector in addition to the input sequence, but we won't detail that here.

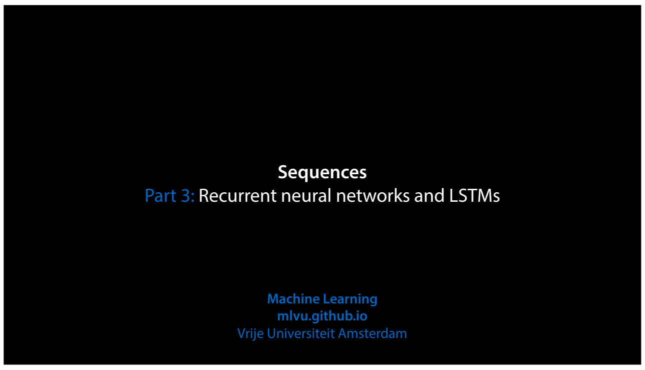


recap

Deep learning on sequences:

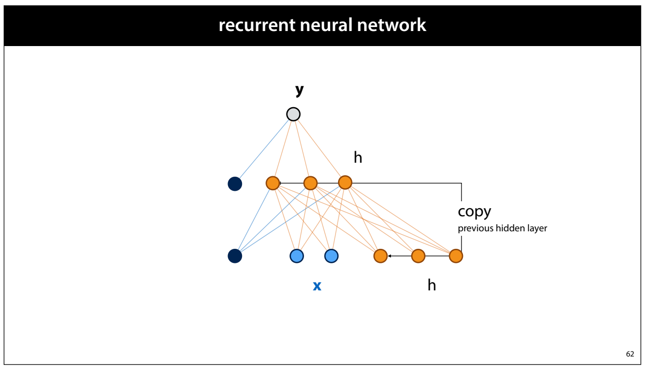
- input: sequence of vectors
- layers: sequence-to-sequence
- output: global pooling or global unit for sequence labelling tasks

60



Sequences
Part 3: Recurrent neural networks and LSTMs

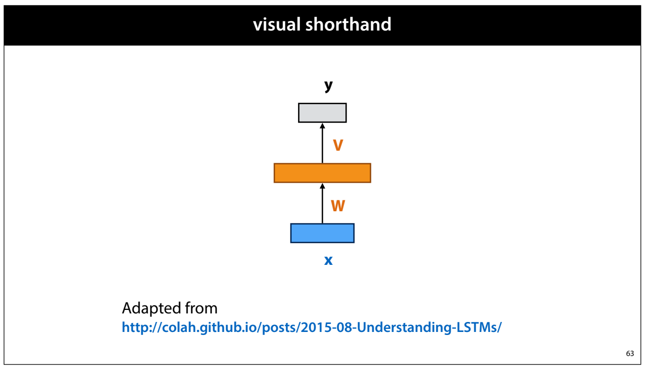
Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam



recurrent neural network

One of the most popular neural networks for sequences is the **recurrent neural network**. This is a generic name for any neural network with cycles in it.

The figure shows a popular configuration. It's a basic fully connected network, except that its input x is extended by three nodes to which the hidden layer is copied.

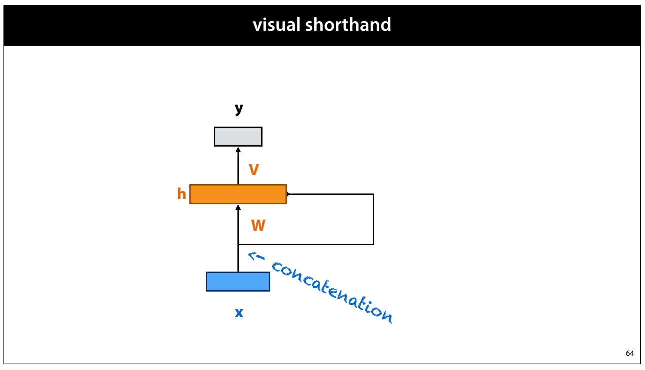


visual shorthand

Adapted from
http://colah.github.io/posts/2015-08-Understanding-LSTMs/

To keep things clear we will adopt this visual shorthand: a rectangle represents a vector of nodes, and an arrow feeding into such a rectangle annotated with a weight matrix represents a fully connected transformation.

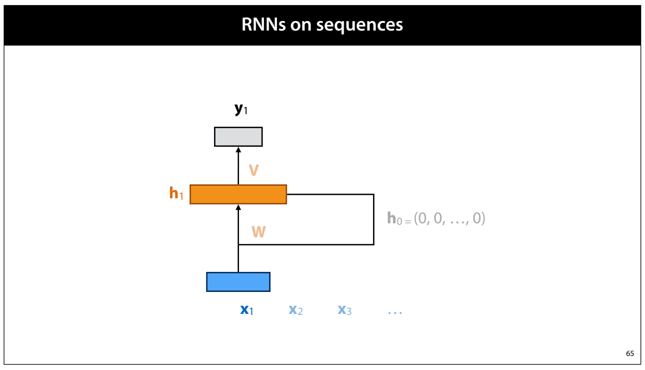
We will assume bias nodes are included without drawing them.



visual shorthand

A line with no weight matrix represents a copy of the input vector. When two lines flow into each other, we concatenate their vectors.

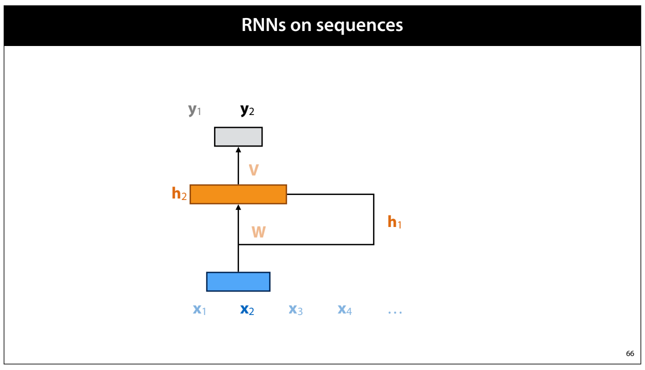
Here, the added line copies h, concatenates it to x, and applies weight matrix W.



RNNs on sequences

We can now apply this neural network to a sequence. We feed it the first input, x_1, result in a first value for the hidden layer, h_1, and retrieve the first output y_1.

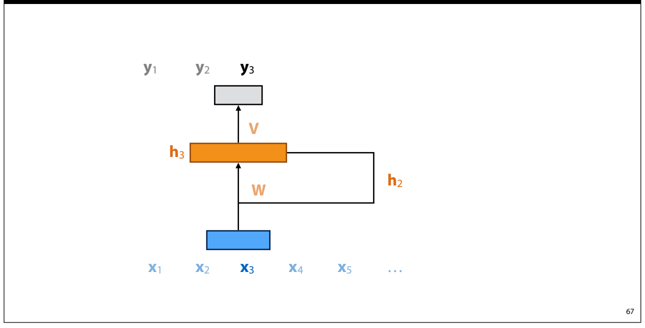
The hidden nodes are initialise to zero, so at first the network behaves just like a fully connected network.



RNNs on sequences

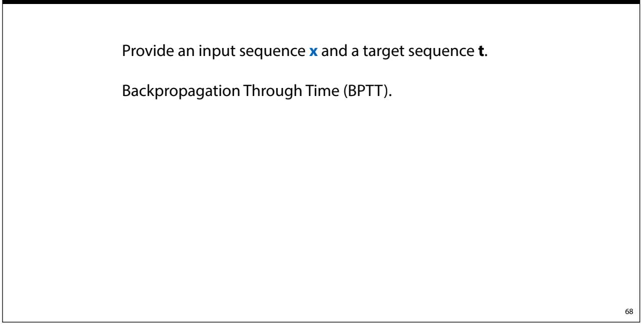
We then feed the second input in the sequence, x_1. We now receive the *previous* hidden layer, h_1, concatenate it to the input, and multiply it by W, to produce the second hidden layer h_2. This is multiplied by V to produce the second output.

RNNs on sequences



And so on.

how to train RRNs?

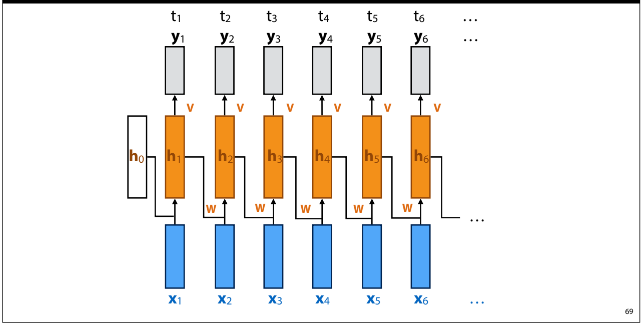

Provide an input sequence **x** and a target sequence **t**.

Backpropagation Through Time (BPTT).

68

In principle, we

how to train RNNs? unrolling

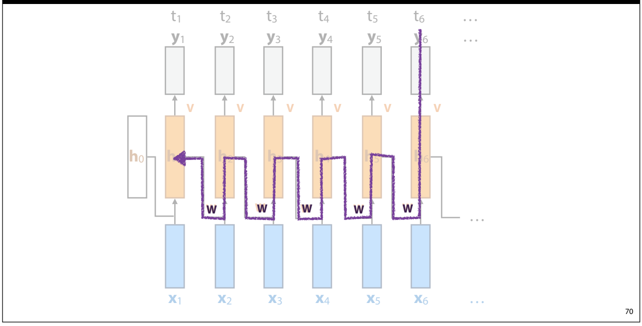


Instead of visualising a single small network, applied at every time step, we can **unroll** the network. Every step in the sequence is applied in parallel to a copy of the network, and the recurrent connection flows from the previous copy to the next.

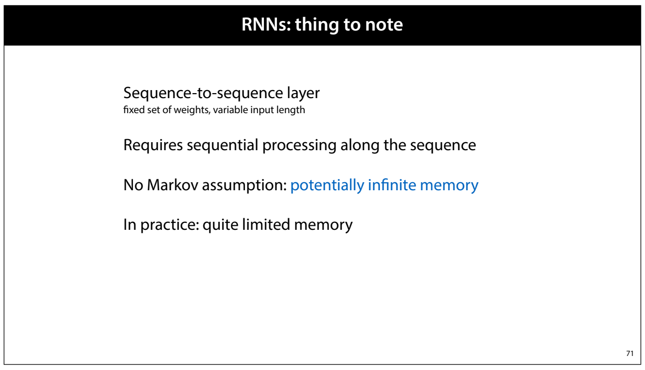
Now the whole network is just one big, complicated **feedforward net**, that is, a network *without* cycles. Note that we have a lot of shared weights, but we know how to deal with those.

The hidden layer is initialised to the zero vector.

backpropagation *through time*



Now the whole network is just one big, complicated feedforward net. Note that we have a lot of shared weights, but we know how to deal with those.



RNNs: thing to note

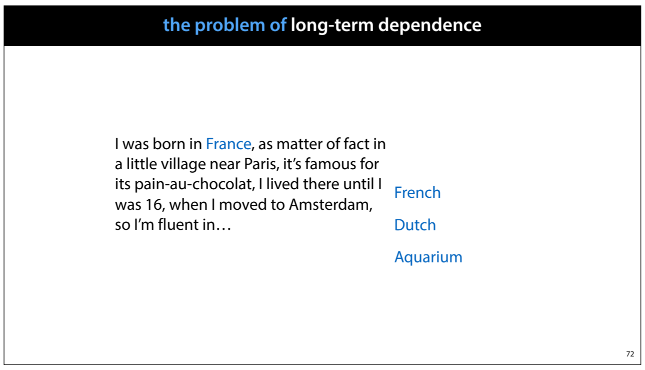
Sequence-to-sequence layer
fixed set of weights, variable input length

Requires sequential processing along the sequence

No Markov assumption: potentially infinite memory

In practice: quite limited memory

71



the problem of long-term dependence

I was born in France, as matter of fact in a little village near Paris, it's famous for its pain-au-chocolat, I lived there until I was 16, when I moved to Amsterdam, so I'm fluent in…

French

Dutch

Aquarium

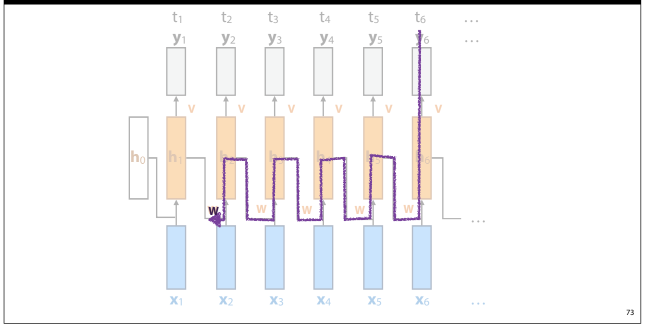
72

Basic RNNs work pretty well, but they do not learn to remember information for very long. Technically they can, but the gradient vanished too quickly over the timesteps.

You can't have a long term memory for everything. You need to be selective, and you need to learn to select words to be stored for the long term when you first see them.

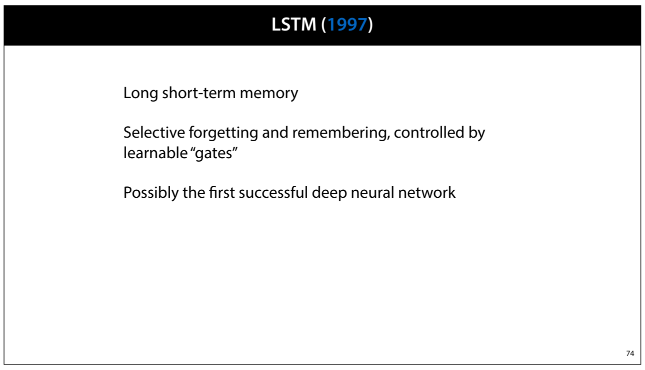
In order to remember things long term you need to forget many other things.

vanishing gradients



One of the reasons that neural networks don remember too well is that the weights between an input long ago and the current output get a gradient that has to travel through a lot of layers in our network. If these are sigmoid activated layers, these gradients will vanish much more strongly than the gradients between input 6 and output 6, so the network will always learn more from short term correlations between the input and the output than from long term correlations.

We could fix this with ReLU activations and other tricks, but in the late 1990s most of this tricks weren't available yet. Instead the LSTM was invented, which solved this problem, by moving the activations out of the way of the gradient travelling backwards in time.



LSTM (1997)

Long short-term memory

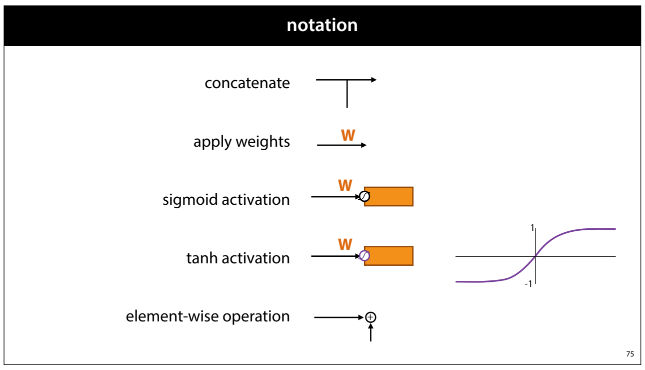
Selective forgetting and remembering, controlled by learnable "gates"

Possibly the first successful deep neural network

74

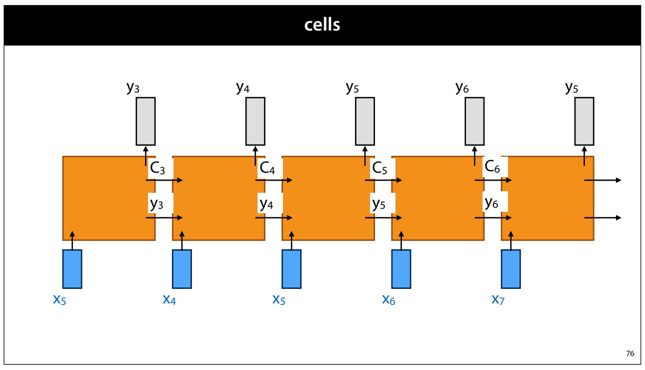
An enduring solution to the problem are LSTMs. LSTMs have a complex mechanism, which we'll go through step by step. To do so, we'll first set up a visual notation.

notation

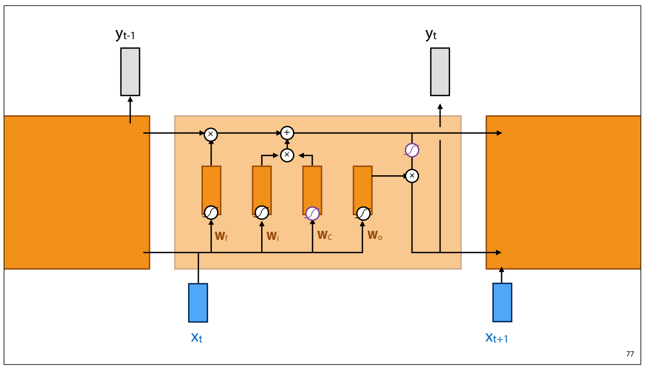


Here is our visual notation.

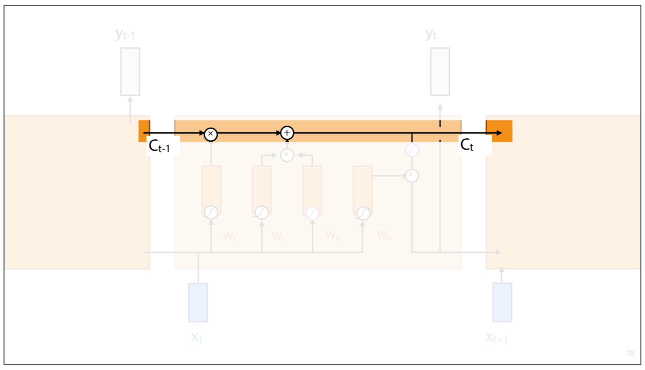
cells



The basic operation of the LSTM is called a cell (the orange square, which we'll detail later). Between cells, there are two recurrent connections, y, the current output, and C the **cell state**.

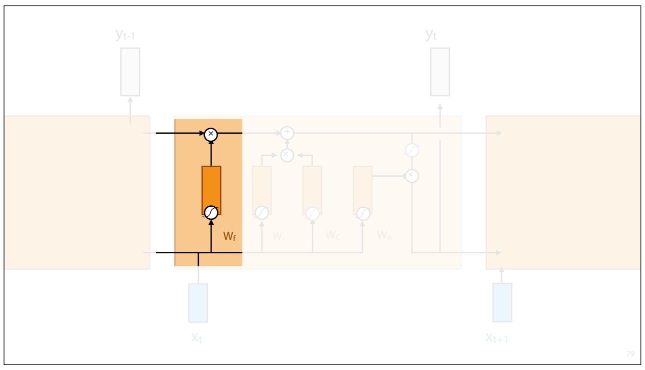


Here is what happens inside the cell. It looks complicated, but we'll go through all the elements step by step.



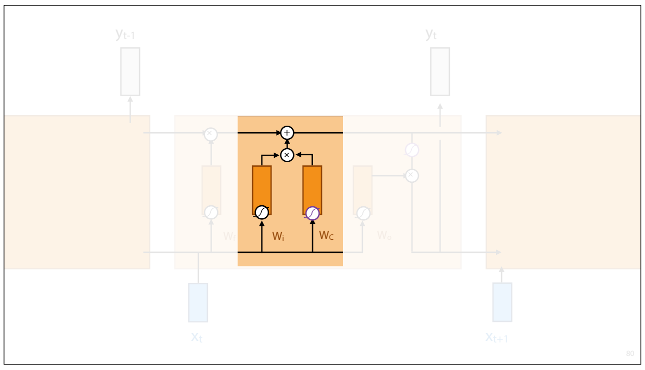
The first is the "conveyor belt". It passes the previous cell state to the next cell. Along the way, the current input can be used to manipulate it.

Note that the connection from the previous cell to the next has *no activations*. This means that along this path, gradients do not decay. It's also very easy for an LSTM cell to ignore the current information and just pass the information along the conveyor belt.

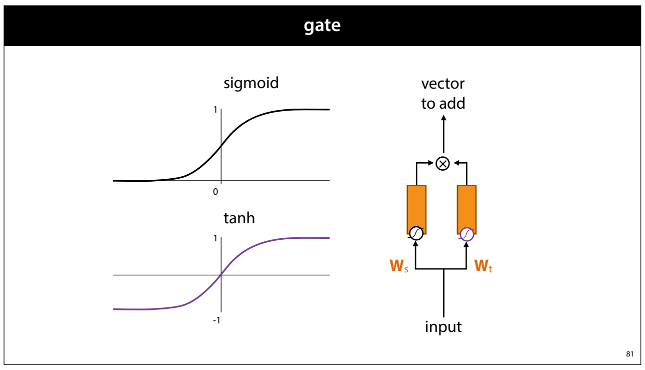


Here is the first manipulation of the conveyor belt. This is called the **forget gate**.

It looks at the current input, concatenated with the previous output, and applies an element-wise scaling to the current value in the conveyor belt. Outputting all 1s will keep the current value on the belt what it is, and outputting all values near 0, will decay the values (forgetting what we've seen so far, and allowing it to be replaces by our new values in the next step).



in the next step, we pass the

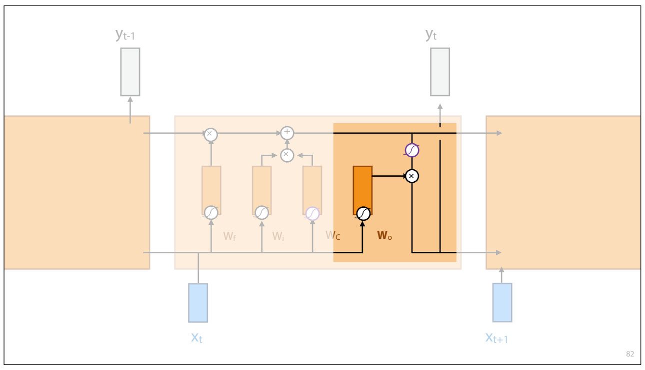


The gating mechanism a single input vector, projects it to two different vectors, one using a sigmoid and one using a a tanh activation.

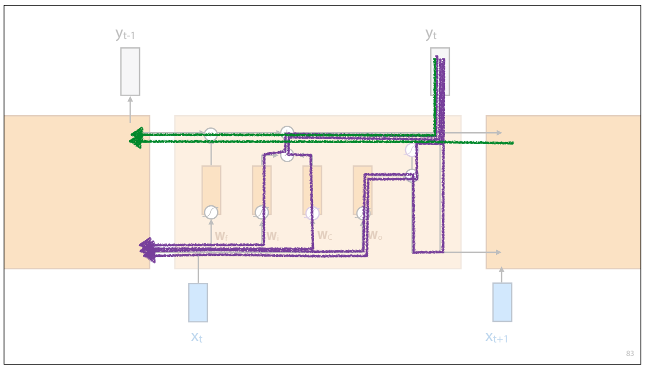
The gate is best understand as producing an additive value: we want to figure out how much of the input to add to some other vector, in this case the one on the converyor belt. If the current input is important we want to add most of it, at the risk of forgetting what we have in memory, and it the current input is unimportant, we want to ignore it.

The tanh should be though of as a mapping of the input to the range [-1, 1]. This is the value we will add to the conveyor belt.

The sigmoid acts as a selection or attention vector. For elements of the input that are important, it outputs 1, retaining all the input in the addition vector. For elements of the input that are not important, it outputs 0, so that they are zeroed out. The sigmoid and tanh vectors are element-wise multiplied.



Finally, we need to decide what to output now. We take the current value of the conveyor belt, tanh it to rescale, and element-wise multiply it by another sigmoid activated layer. This layer is sent out as the current output, and sent to the next cell along the second recurrent connection.



Now, when we look at all the paths that the gradients take back down the network, we see that there are many of them, in purple that cross activations, and these will likely die out over a few cells, and never contribute to learning long range dependencies.

However, there are also paths for the gradient, in green, that travel over the conveyor belt and only encounter linear operations. This means that these gradients are perfectly preserved as they travel back through time, and can be used to pick up on long range dependencies. This is where the name comes from. The conveyor belt functions as a long term memory preserving good gradients, and the rest of the functions as a short term memory, making nonlinear projections of the input and using them to manipulate the output and the contents of the conveyor belt.

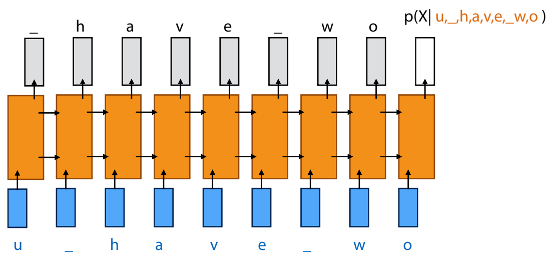
LSTM language model

p(prize | a, won, have, you, congratulations)

NB: no Markov assumption!

That's a lot of complexity. Let's see what it buys us. The best way, by far, to illustrate the power of LSTMs is to apply the sequential sampling trick.

character level language model



So, now that we have a powerful, recurrent sequence-to-sequence layer, let's see what it can do. We'll train a languange model autoregressively, as we did with the markov model, but this time we'll train it at character-level.

We cut a corpus of text into large chunks of character sequences, and we feed these to our model, teaching it to predict the next character at each position in the sequence.

sequential *sampling* from a language model

start with a small *seed* sequence s = [c_1, c_2, c_3] of tokens.

loop:

Sample next char c according to p(C = c | c_1, c_2, …)
feed the *whole* seed to the network

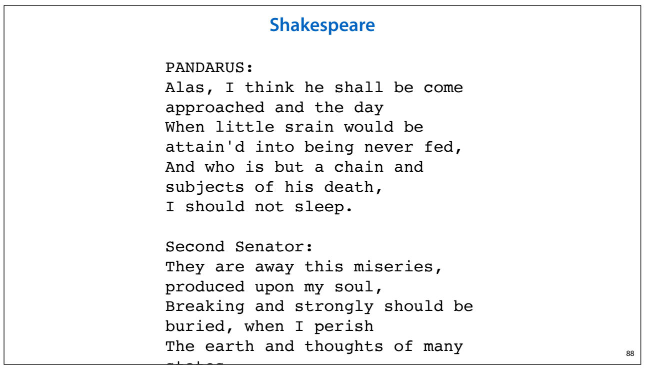
append c to s

After training, we start with a small seed of characters, and we sample sequentially. Note that this time we have no Markov assumption, so we keep feeding the whole sequence to the network very time we sample.

some examples

source: The Unreasonable Effectiveness of Recurrent Neural Networks
Andrej Karpathy

http://karpathy.github.io/2015/05/21/rnn-effectiveness/



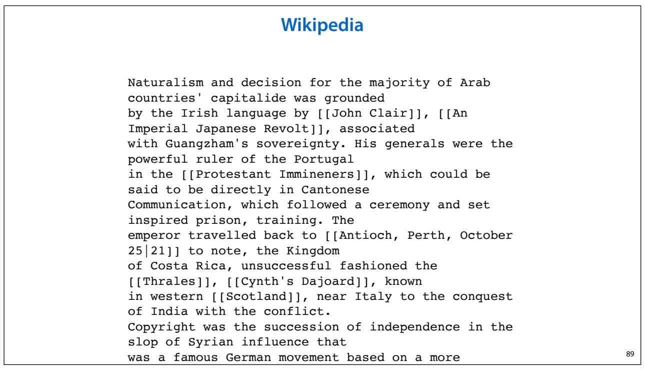
Shakespeare

```
PANDARUS:
Alas, I think he shall be come
approached and the day
When little srain would be
attain'd into being never fed,
And who is but a chain and
subjects of his death,
I should not sleep.

Second Senator:
They are away this miseries,
produced upon my soul,
Breaking and strongly should be
buried, when I perish
The earth and thoughts of many
```

88

Remember, this is a **character level** language model.

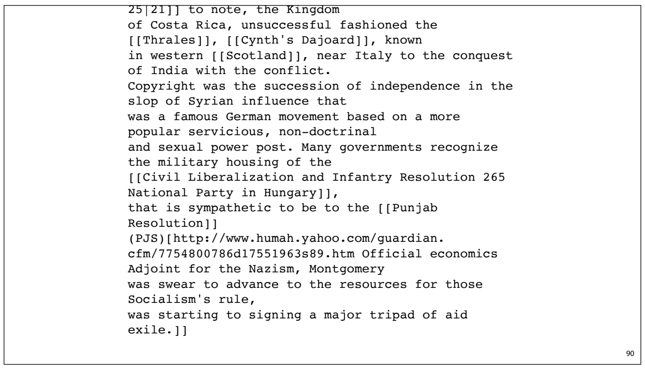


Wikipedia

```
Naturalism and decision for the majority of Arab
countries' capitalide was grounded
by the Irish language by [[John Clair]], [[An
Imperial Japanese Revolt]], associated
with Guangzham's sovereignty. His generals were the
powerful ruler of the Portugal
in the [[Protestant Immineners]], which could be
said to be directly in Cantonese
Communication, which followed a ceremony and set
inspired prison, training. The
emperor travelled back to [[Antioch, Perth, October
25|21]] to note, the Kingdom
of Costa Rica, unsuccessful fashioned the
[[Thrales]], [[Cynth's Dajoard]], known
in western [[Scotland]], near Italy to the conquest
of India with the conflict.
Copyright was the succession of independence in the
slop of Syrian influence that
was a famous German movement based on a more
```

89

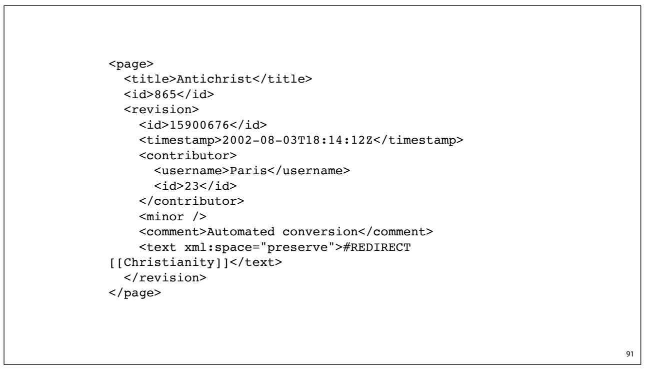
Note that not only is the language natural, the wikipedia markup is also correct (link brackets are closed properly, and contain key concepts).



```
25|21]] to note, the Kingdom
of Costa Rica, unsuccessful fashioned the
[[Thrales]], [[Cynth's Dajoard]], known
in western [[Scotland]], near Italy to the conquest
of India with the conflict.
Copyright was the succession of independence in the
slop of Syrian influence that
was a famous German movement based on a more
popular servicious, non-doctrinal
and sexual power post. Many governments recognize
the military housing of the
[[Civil Liberalization and Infantry Resolution 265
National Party in Hungary]],
that is sympathetic to be to the [[Punjab
Resolution]]
(PJS)[http://www.humah.yahoo.com/guardian.
cfm/7754800786d17551963s89.htm Official economics
Adjoint for the Nazism, Montgomery
was swear to advance to the resources for those
Socialism's rule,
was starting to signing a major tripad of aid
exile.]]
```

90

The network can even learn to generate valid (looking) URLs for external links.

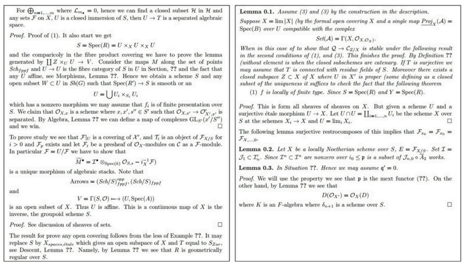


```
<page>
  <title>Antichrist</title>
  <id>865</id>
  <revision>
    <id>15900676</id>
    <timestamp>2002-08-03T18:14:12Z</timestamp>
    <contributor>
      <username>Paris</username>
      <id>23</id>
    </contributor>
    <minor />
    <comment>Automated conversion</comment>
    <text xml:space="preserve">#REDIRECT
[[Christianity]]</text>
  </revision>
</page>
```

91

Sometimes wikipedia text contains bits of XML for structured information. The model can generate these flawlessly.

LaTeX



For $\bigoplus_{n=1,\ldots,m}$ where $\mathcal{L}_{m_\bullet} = 0$, hence we can find a closed subset $\mathcal{H}$ in $\mathcal{H}$ and any sets $\mathcal{F}$ on X, U is a closed immersion of S, then $U \to T$ is a separated algebraic space.

Proof. Proof of (1). It also start we get

$$S = \mathrm{Spec}(R) = U \times_X U \times_X U$$

and the comparicoly in the fibre product covering we have to prove the lemma generated by $\coprod Z \times_U U \to V$. Consider the maps M along the set of points Sch_{fppf} and $U \to U$ is the fibre category of S in U in Section, ?? and the fact that any U affine, see Morphisms, Lemma ??. Hence we obtain a scheme S and any open subset $W \subset U$ in $Sh(G)$ such that $\mathrm{Spec}(R') \to S$ is smooth or an

$$U = \bigcup U_i \times_{S_i} U_i$$

which has a nonzero morphism we may assume that f_i is of finite presentation over S. We claim that $\mathcal{O}_{X,x}$ is a scheme where $x, x', s'' \in S'$ such that $\mathcal{O}_{X,x'} \to \mathcal{O}'_{X',x'}$ is separated. By Algebra, Lemma ?? we can define a map of complexes $\mathrm{GL}_{S'}(x'/S'')$ and we win. □

To prove study we see that $\mathcal{F}|_U$ is a covering of $\mathcal{X}'$, and $\mathcal{T}_i$ is an object of $\mathcal{F}_{X/S}$ for $i > 0$ and $\mathcal{F}_p$ exists and let $\mathcal{F}_i$ be a presheaf of $\mathcal{O}_X$-modules on $\mathcal{C}$ as a $\mathcal{F}$-module. In particular $\mathcal{F} = U/\mathcal{F}$ we have to show that

$$\widetilde{M}^\bullet = \mathcal{I}^\bullet \otimes_{\mathrm{Spec}(k)} \mathcal{O}_{S,s} - i_X^{-1}\mathcal{F})$$

is a unique morphism of algebraic stacks. Note that

$$\mathrm{Arrows} = (Sch/S)^{opp}_{fppf}, (Sch/S)_{fppf}$$

and

$$V = \Gamma(S, \mathcal{O}) \longmapsto (U, \mathrm{Spec}(A))$$

is an open subset of X. Thus U is affine. This is a continuous map of X is the inverse, the groupoid scheme S.

Proof. See discussion of sheaves of sets. □

The result for prove any open covering follows from the less of Example ??. It may replace S by $X_{spaces,étale}$ which gives an open subspace of X and T equal to S_{Zar}, see Descent, Lemma ??. Namely, by Lemma ?? we see that R is geometrically regular over S.

Lemma 0.1. *Assume (3) and (3) by the construction in the description.*

Suppose $X = \lim |X|$ (by the formal open covering X and a single map $\underline{Proj}_X(\mathcal{A}) = \mathrm{Spec}(B)$ over U compatible with the complex

$$Set(\mathcal{A}) = \Gamma(X, \mathcal{O}_{X,\mathcal{O}_X}).$$

When in this case of to show that $\mathcal{Q} \to \mathcal{C}_{Z/X}$ is stable under the following result in the second conditions of (1), and (3). This finishes the proof. By Definition ?? (without element is when the closed subschemes are catenary. If T is surjective we may assume that T is connected with residue fields of S. Moreover there exists a closed subspace $Z \subset X$ of X where U in X' is proper (some defining as a closed subset of the uniqueness it suffices to check the fact that the following theorem

(1) *f is locally of finite type. Since $S = \mathrm{Spec}(R)$ and $Y = \mathrm{Spec}(R)$.*

Proof. This is form all sheaves of sheaves on X. But given a scheme U and a surjective étale morphism $U \to X$. Let $U \cap U = \coprod_{i=1,\ldots,n} U_i$ be the scheme X over S at the schemes $X_i \to X$ and $U = \lim_i X_i$. □

The following lemma surjective restrocomposes of this implies that $\mathcal{F}_{x_0} = \mathcal{F}_{x_0} = \mathcal{F}_{X,\ldots,0}$.

Lemma 0.2. *Let X be a locally Noetherian scheme over S, $E = \mathcal{F}_{X/S}$. Set $\mathcal{I} = \mathcal{J}_1 \subset \mathcal{I}'_n$. Since $\mathcal{I}^n \subset \mathcal{I}^n$ are nonzero over $i_0 \leq \mathfrak{p}$ is a subset of $\mathcal{J}_{n,0} \circ \overline{A}_2$ works.*

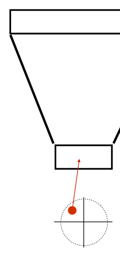
Lemma 0.3. *In Situation ??. Hence we may assume $\mathfrak{q}' = 0$.*

Proof. We will use the property we see that $\mathfrak{p}$ is the mext functor (??). On the other hand, by Lemma ?? we see that

$$D(\mathcal{O}_{X'}) = \mathcal{O}_X(D)$$

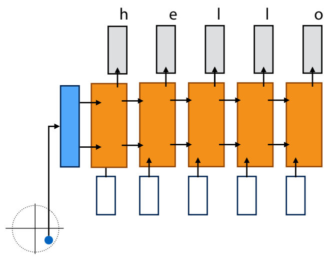
where K is an F-algebra where δ_{n+1} is a scheme over S. □

generative sequence model



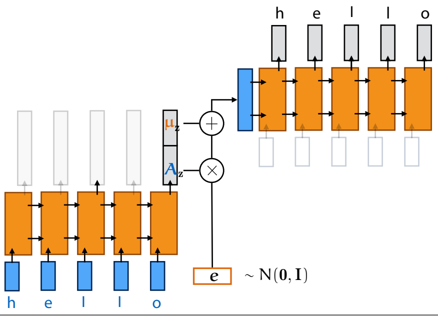
Another way to train a model to generate sequences is to use the generator network that we saw previously. We sample a *single* vector from a standard normal distribution and feed it to a sequence-to-sequence network

generative sequence model



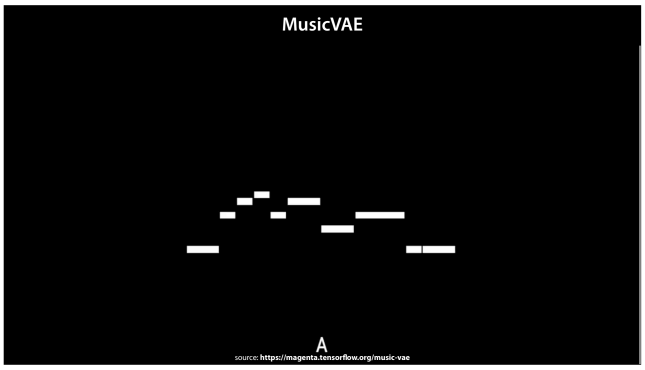
In the previous video we saw that we could achieve this by repeating the input vector into a sequence, but when we use RNNs, there is another option. We can feed the latent vector to the network as the initial hidden state. We canthen set the input sequence to zero, or as we will see later, use it for something else.

sequence to sequence (variational) AE



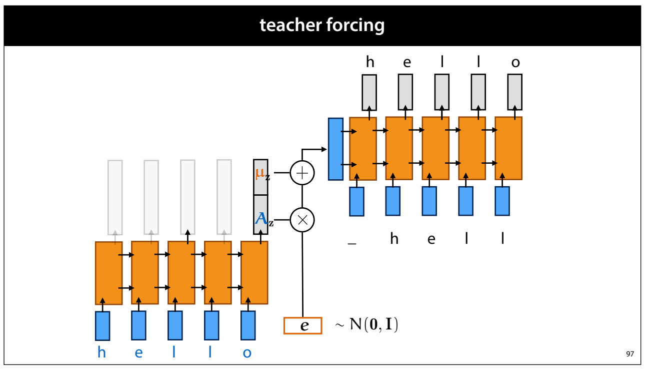
Of course, if we want to train a generator network we'll need either a discriminator or an encoder.

For an autoencoder we can simply use a sequence-to-label network as our encoder, and interpret the last vector of the output as the latent vector representing the whole input.


MusicVAE

source: https://magenta.tensorflow.org/music-vae

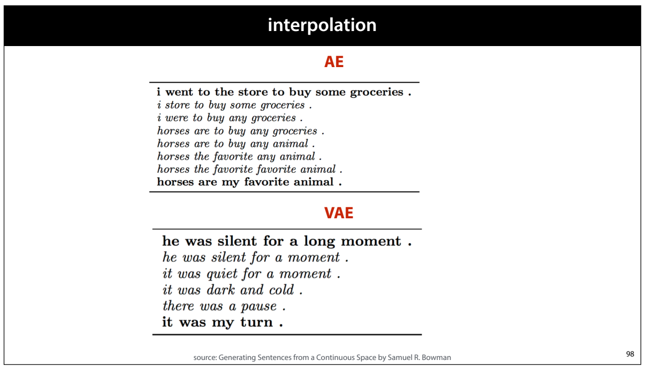
source: **https://magenta.tensorflow.org/music-vae**


teacher forcing

This works well for music but we’ve lost the fine touch of the sequential sampling, where we could build up a sentence character by character, finetuning the details one at a time.

Instead a single small latent vector needs to represent all information in the sequence: this is great to capture the global pattern, but not for deciding on the finer details. This is similar to what we saw with images: the VAE reconstructions we recognizable people but the finer details of the image had been washed out.

Here, we can combine the best of both worlds. Since we’d left the input sequence blank, we can feed the decoder a character-shifted version of the input as well as the latent vector. This way, we are training a decoder that gets global information from the latent vector, and generates the finer details by autoregressive sampling.


interpolation

AE

i went to the store to buy some groceries .
i store to buy some groceries .
i were to buy any groceries .
horses are to buy any groceries .
horses are to buy any animal .
horses the favorite any animal .
horses the favorite favorite animal .
horses are my favorite animal .

VAE

he was silent for a long moment .
he was silent for a moment .
it was quiet for a moment .
it was dark and cold .
there was a pause .
it was my turn .

source: Generating Sentences from a Continuous Space by Samuel R. Bowman

The example we saw in the autoencoder lecture was also trained in this way: using a variational autoencoder with teacher forcing.

source: Generating Sentences from a Continuous Space by Samuel R. Bowman, Luke Vilnis, Oriol Vinyals, Andrew M. Dai, Rafal Jozefowicz, Samy Bengio https://arxiv.org/abs/1511.06349

sketch-rnn

source: **A Neural Representation of Sketch Drawings**, Ha and Eck (2017)

99

Here is another sequence-to-sequence autoencoder. To gather data, Google set up a website, where it challenged users to quickly draw a sketch of something using their mouse. The drawing was captured as a *sequence* of points in the plane.

Researchers then trained a variational sequence-to-sequence autoencoder with teacher forcing and a mixture density output at each step of the sequence. In short, quite a complex model. The result, however was worth it. Here, we see two interpolation grids from the latent space of this model, smoothly interpolating between vairous ways of drawing cats and various ways of drawing owls.

more detail: dlvu.github.io (lectures 5 and 12)

mlcourse@peterbloem.nl

summary

Sequence modeling: we can use existing methods through feature extraction, but be careful about validation.

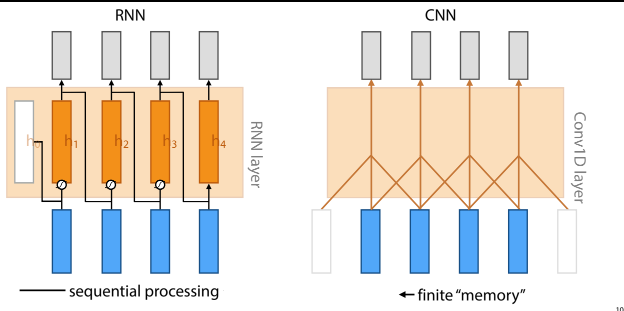
Markov models: simple, finite-memory sequence modeling
For classification or generation

Word2Vec: word embeddings reflecting similarity, semantics

LSTMs: incredibly powerful language models. Tricky to train, very opaque.

101

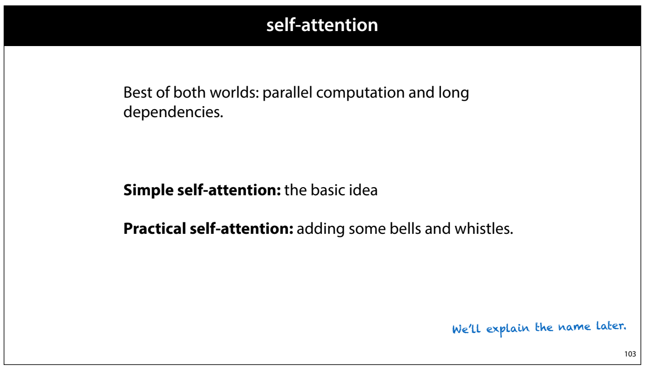
recurrent connections, convolutions



102

We've seen two examples of (non-trivial) sequence-to-sequence layers so far: recurrent neural networks, and convolutions. RNNs have the benefit that they can potentially look infinitely far back into the sequence, but they require fundamentally sequential processing, making them slow. Convolution don't have this drawback—we can compute each output vector in parallel if we want to—but the downside is that they are limited in how far back they can look into the sequence.

Self-attention is another sequence-to-sequence layer, and one which provides us with the best of both worlds: parallel processing and a potentially infinite memory.



self-attention

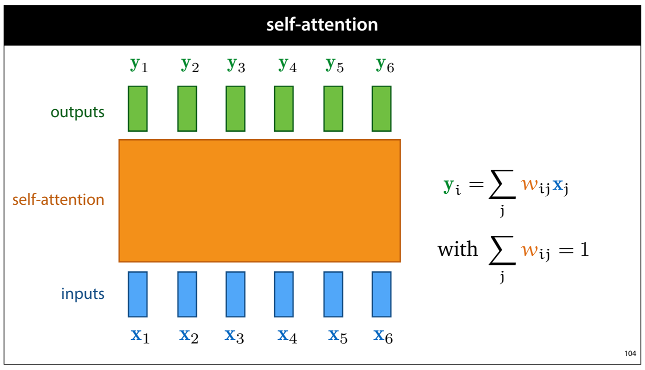
Best of both worlds: parallel computation and long dependencies.

Simple self-attention: the basic idea

Practical self-attention: adding some bells and whistles.

We'll explain the name later.

103



self-attention

y_1 y_2 y_3 y_4 y_5 y_6
outputs
self-attention
inputs
x_1 x_2 x_3 x_4 x_5 x_6

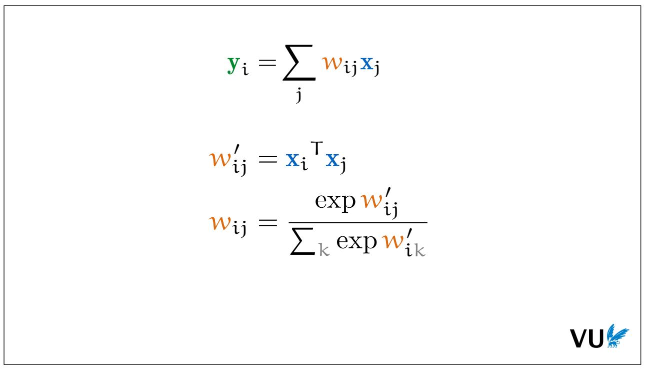
$$y_i = \sum_j w_{ij} x_j$$

$$\text{with} \sum_j w_{ij} = 1$$

104

At heart, the operation of self-attention is very simple. Every output is simply a weighted sum over the inputs. The trick is that the weights in this sum are not parameters. They are derived from the inputs.

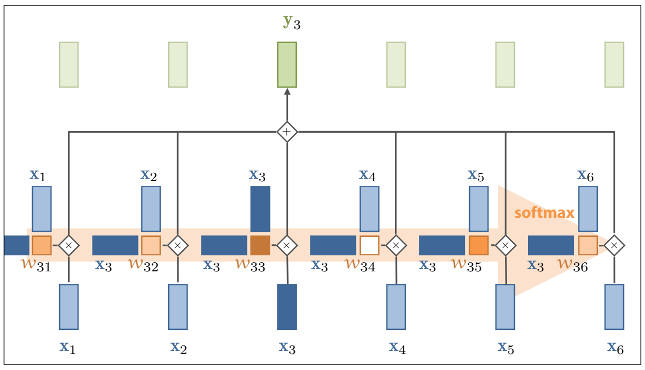
Note that this means that the input and output dimensions of a self-attention layer are always the same. If we want to transform to a different dimension, we'll need to add a projection layer.



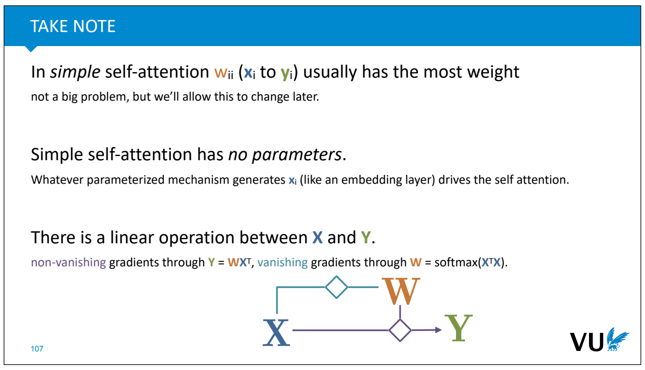
$$y_i = \sum_j w_{ij} x_j$$

$$w'_{ij} = {x_i}^T x_j$$

$$w_{ij} = \frac{\exp w'_{ij}}{\sum_k \exp w'_{ik}}$$



y_3
x_1 x_2 x_3 x_4 x_5 x_6
softmax
w_{31} x_3 w_{32} x_3 w_{33} x_3 w_{34} x_3 w_{35} x_3 w_{36}
x_1 x_2 x_3 x_4 x_5 x_6



TAKE NOTE

In *simple* self-attention w_{ii} (x_i to y_i) usually has the most weight

not a big problem, but we'll allow this to change later.

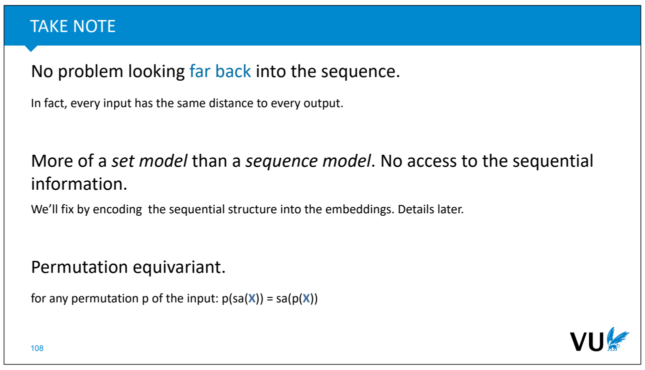
Simple self-attention has *no parameters*.

Whatever parameterized mechanism generates x_i (like an embedding layer) drives the self attention.

There is a linear operation between **X** and **Y**.

non-vanishing gradients through $Y = WX^T$, vanishing gradients through $W = \text{softmax}(X^TX)$.

$$\begin{tikzcd} & W \arrow[d] \\ X \arrow[ur] \arrow[r] & \diamond \arrow[r] & Y \end{tikzcd}$$



TAKE NOTE

No problem looking far back into the sequence.

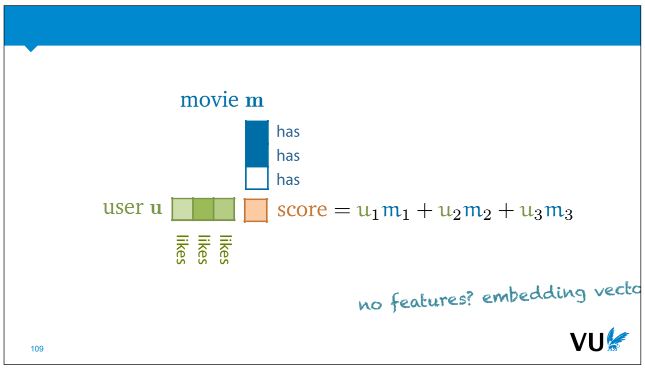
In fact, every input has the same distance to every output.

More of a *set model* than a *sequence model*. No access to the sequential information.

We'll fix by encoding the sequential structure into the embeddings. Details later.

Permutation equivariant.

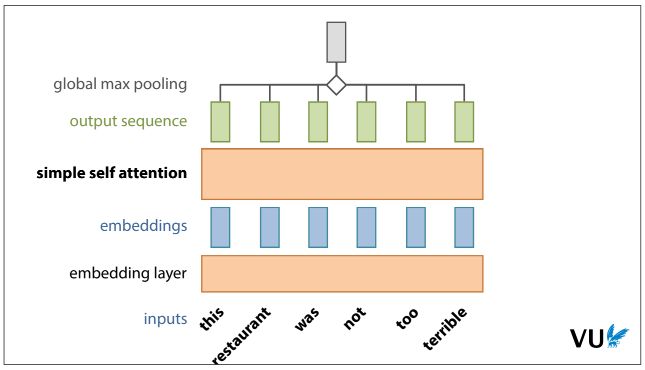
for any permutation p of the input: p(sa(X)) = sa(p(X))



$\text{score} = u_1m_1 + u_2m_2 + u_3m_3$

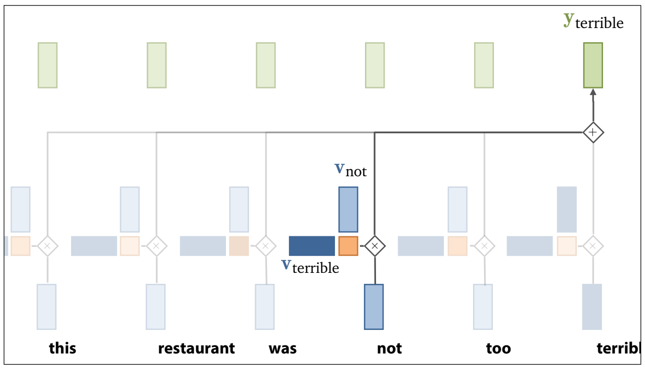
If we had features for each movie and user, we could match them up like this. We multiply how much the user likes romance by how much romance there is in the movie. If both are positive of negative, the score is increased. If one is positive and one is negative, the score is decreased.

Note that we're not just taking into account the sign of the values, but also the magnitude. If a user's preference for action is near zero, it doesn't matter much for the score whether the movie has action.

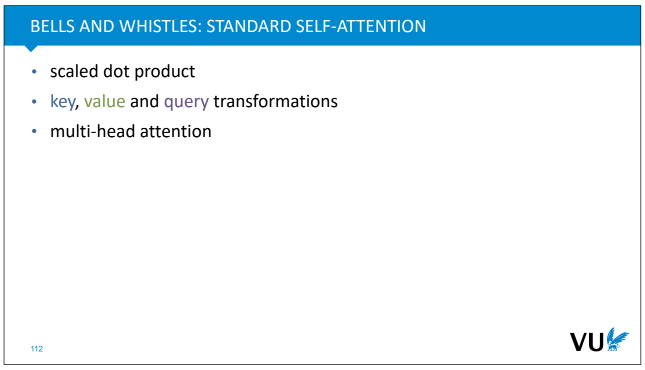


As a simple example, let's build a sequence classifier consisting of just one embedding layer followed by a global maxpooling layer. We'll imagine a sentiment classification task where the aim is to predict whether a restaurant review is positive or negative.

If we did this without the self-attention layer, we would essentially have a model where each word can only contribute to the output score independently of the other. This is known as a bag of words model. In this case, the word terrible would probably cause us to predict that this is a negative review. In order to see that it might be a positive review, we need to recognize that the meaning of the word terrible is moderated by the word not. This is what the self-attention can do for us.



If the embedding vectors of not and terrible have a high dot product together, the weight of the input vector for not becomes high, allowing it to influence the meaning of the word terrible in the output sequence.

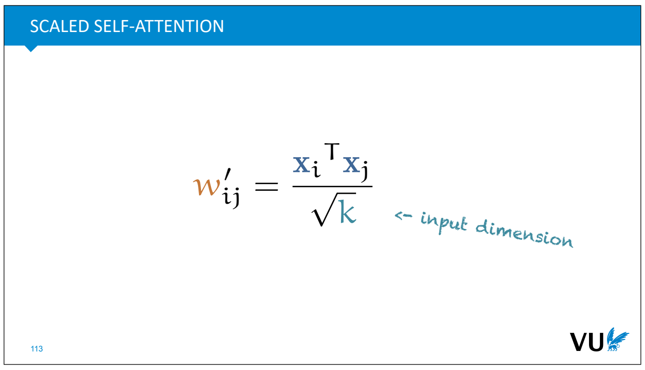


BELLS AND WHISTLES: STANDARD SELF-ATTENTION

- scaled dot product
- key, value and query transformations
- multi-head attention

112

The standard self attention add some bells and whistles to this basic framework. We'll discuss the three most important additions.



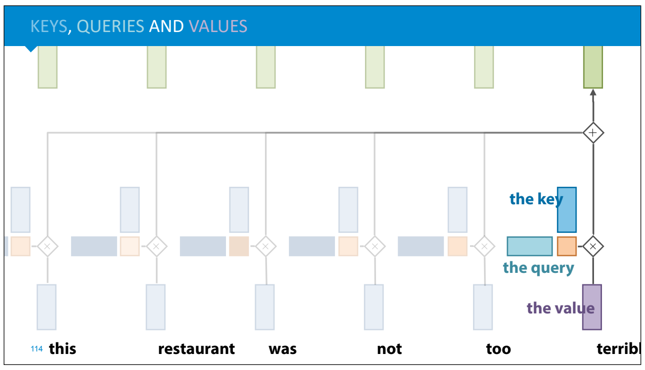
SCALED SELF-ATTENTION

$$w'_{ij} = \frac{x_i^T x_j}{\sqrt{k}}$$

<- input dimension

Scaled self attention is very simple: instead of using the dot product, we use the dot product scaled by the square root of the input dimension. This ensures that the input and output of the self attention operation have similar variance.

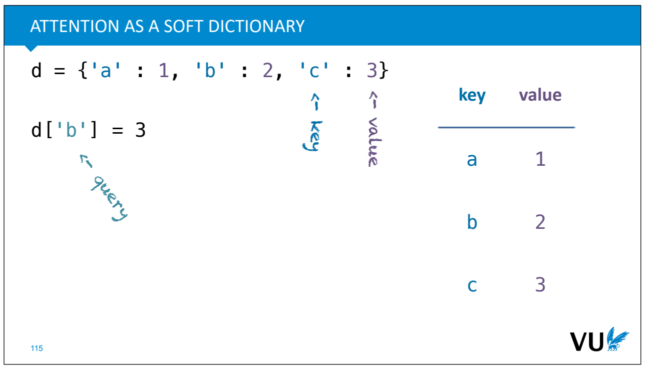
Why √k? Imagine a vector in $\mathbb{R}^k$ with values all c. Its Euclidean length is √kc. Therefore, we are dividing out the amount by which the increase in dimension increases the length of the average vectors. Transformer usually models apply normalization at every layer, so we can usually assume that the input is standard-normally distributed.



KEYS, QUERIES AND VALUES

In each self attention computation, every input vector occurs in three distinct roles:

- **the value**: the vector that is used in the weighted sum that ultimately provides the output
- **the query**: the input vector that corresponds to the current output, matched against every other input vector.
- **the key**: the input vector that the query is matched against to determine the weight.



ATTENTION AS A SOFT DICTIONARY

```
d = {'a' : 1, 'b' : 2, 'c' : 3}

d['b'] = 3
```

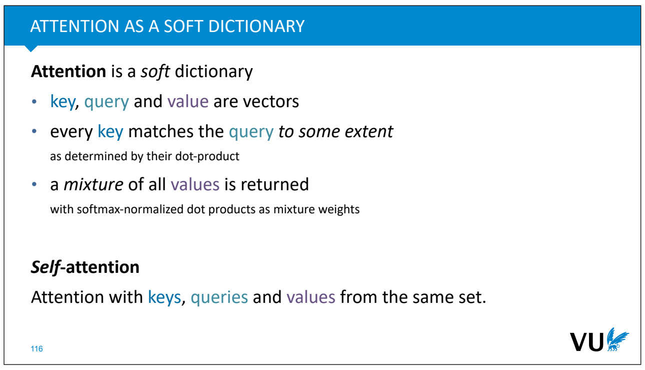
<- key

<- value

<- query

key	value
a	1
b	2
c	3

In a dictionary, all the operations are discrete: a query only matches a single key, and returns only the value corresponding to that key.



ATTENTION AS A SOFT DICTIONARY

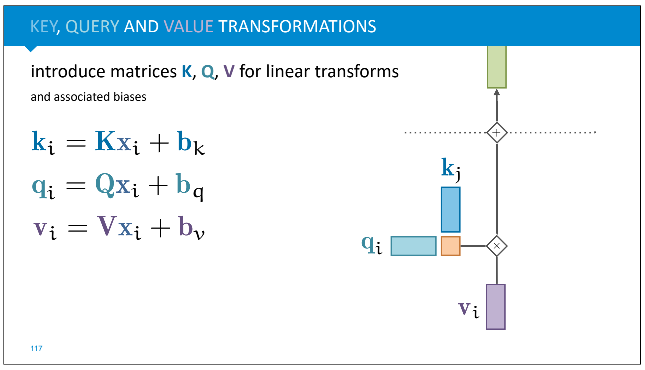
Attention is a *soft* dictionary

- key, query and value are vectors
- every key matches the query *to some extent*
  as determined by their dot-product
- a *mixture* of all values is returned
  with softmax-normalized dot products as mixture weights

***Self*-attention**

Attention with keys, queries and values from the same set.

If the dot product of only one query/key pair is non-zero, we recover the operation of a normal dictionary.



KEY, QUERY AND VALUE TRANSFORMATIONS

introduce matrices **K**, **Q**, **V** for linear transforms

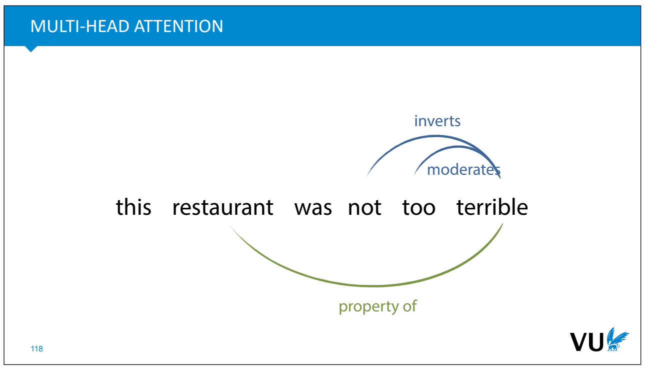
and associated biases

$$\mathbf{k}_i = \mathbf{K}\mathbf{x}_i + \mathbf{b}_k$$
$$\mathbf{q}_i = \mathbf{Q}\mathbf{x}_i + \mathbf{b}_q$$
$$\mathbf{v}_i = \mathbf{V}\mathbf{x}_i + \mathbf{b}_v$$

$\mathbf{k}_j$
$\mathbf{q}_i$
$\mathbf{v}_i$

To give the self attention some more flexibility in determining its behavior, we multiply each input vector by three different k-by-k parameter matrices, which gives us a different vector to act as key query and value.

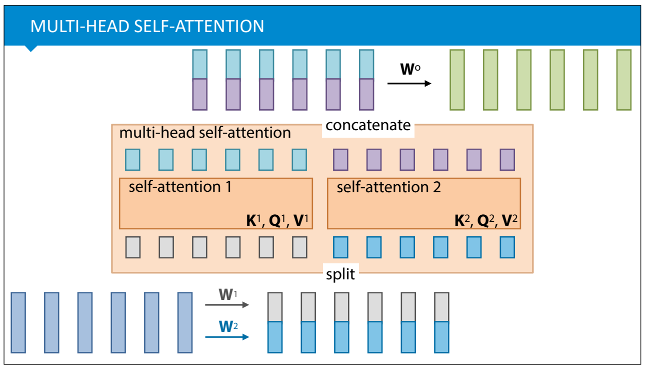
Note that this makes the self attention operation a layer with parameters (where before it had none).



MULTI-HEAD ATTENTION

In many sentences, there are different relations to model. Here, the word meaning of the word "terrible" is inverted by "not" and moderated by "too". Its relation to the word restaurant is completely different: it describes a property of the restaurant.

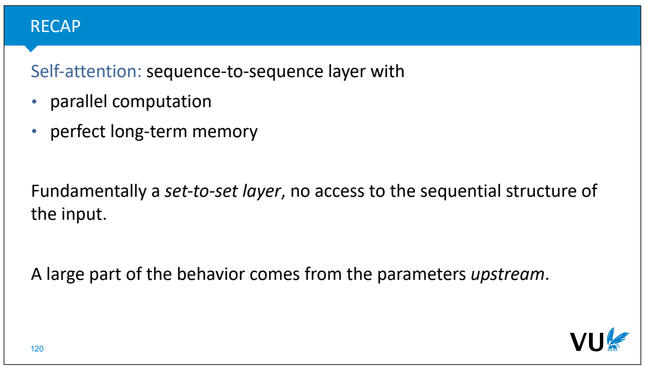
The idea behind multi-head self-attention is that multiple relations are best captured by different self-attention operations.



MULTI-HEAD SELF-ATTENTION

$\mathbf{W}^o$
multi-head self-attention
concatenate
self-attention 1
$\mathbf{K}^1, \mathbf{Q}^1, \mathbf{V}^1$
self-attention 2
$\mathbf{K}^2, \mathbf{Q}^2, \mathbf{V}^2$
split
$\mathbf{W}^1$
$\mathbf{W}^2$

The idea of multi-head attention, is that we project the input sequence down to several lower dimensional sequences, and apply a separate low-dimensional self attention to each of these. After this, we concatenate their outputs, and apply another linear transformation (biases not shown)



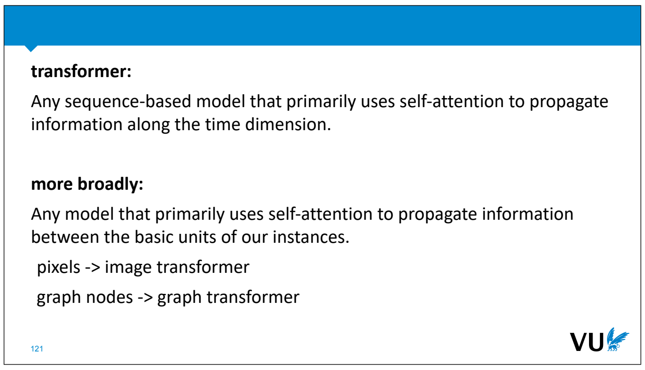
RECAP

Self-attention: sequence-to-sequence layer with

- parallel computation
- perfect long-term memory

Fundamentally a *set-to-set layer*, no access to the sequential structure of the input.

A large part of the behavior comes from the parameters *upstream*.



transformer:

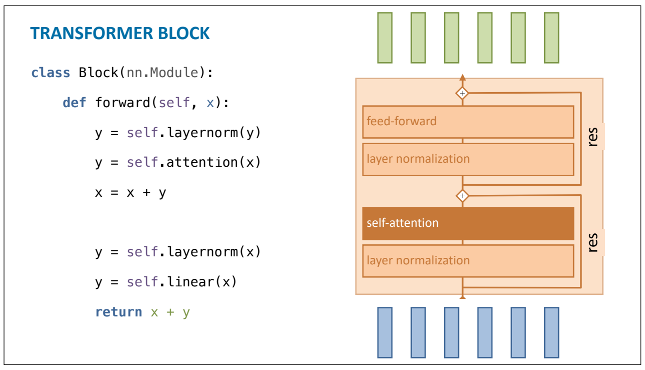
Any sequence-based model that primarily uses self-attention to propagate information along the time dimension.

more broadly:

Any model that primarily uses self-attention to propagate information between the basic units of our instances.

pixels -> image transformer

graph nodes -> graph transformer



TRANSFORMER BLOCK

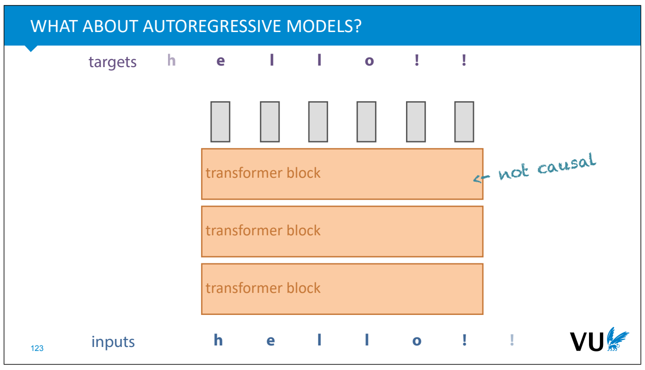
```
class Block(nn.Module):
    def forward(self, x):
        y = self.layernorm(y)
        y = self.attention(x)
        x = x + y

        y = self.layernorm(x)
        y = self.linear(x)
        return x + y
```

The basic building block of transformer models is usually a simple **transformer block**.

The details differ per transformer, but the basic ingredients are usually: one self-attention, one feed-forward layer applied individually to each token in the sequence and a layer normalization and residual connection for each.

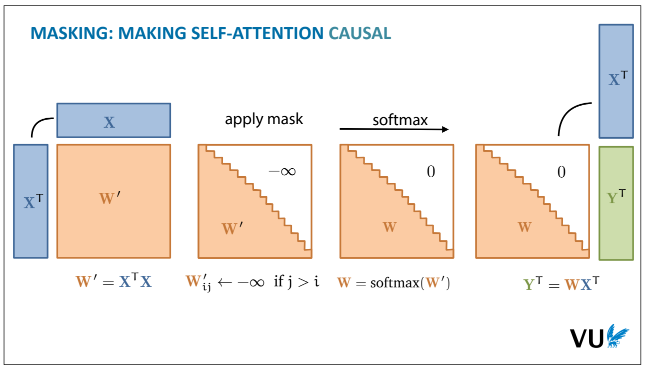
Note that the self-attention is the only operation in the block that propagates information across the time dimension. The other layers operation only on each token independently.s



WHAT ABOUT AUTOREGRESSIVE MODELS?

What about autoregressive modeling?

If we do this naively, we have a problem: the self-attention operation can just look ahead in the sequence to predict what the next model will be. We will never learn to predict the future from the past. In short the transformer block is not a *causal* sequence-to-sequence operation.

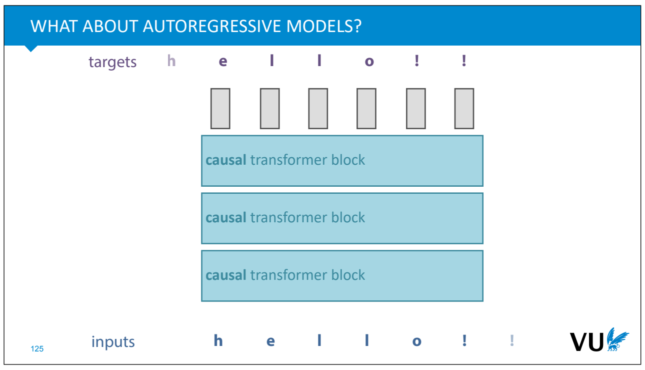


MASKING: MAKING SELF-ATTENTION CAUSAL

apply mask
softmax
$\mathbf{X}$
$\mathbf{X}^T$
$\mathbf{W}'$
$-\infty$
0
$\mathbf{W}$
$\mathbf{Y}^T$
$\mathbf{W}' = \mathbf{X}^T\mathbf{X}$
$\mathbf{W}'_{ij} \leftarrow -\infty \text{ if } j > i$
$\mathbf{W} = \text{softmax}(\mathbf{W}')$
$\mathbf{Y}^T = \mathbf{W}\mathbf{X}^T$

The solution is simple: when we compute the attention weights, we mask out any attention from the current token to future tokens in the sequence.

Note that to do this, we need to set the raw attention weights to negative infinity, so that after the softmax operation, they become 0.

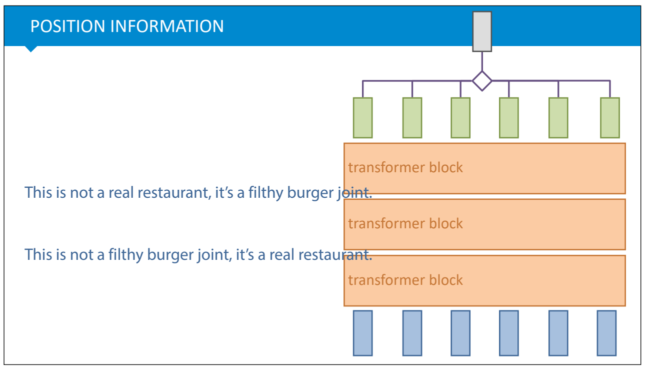


WHAT ABOUT AUTOREGRESSIVE MODELS?

125

Since the self attention is the only part of the transformer block that propagates information across the time dimension, making that part causal, makes the whole block causal.

With a stack of causal transformer blocks, we can easily build an autoregressive model.



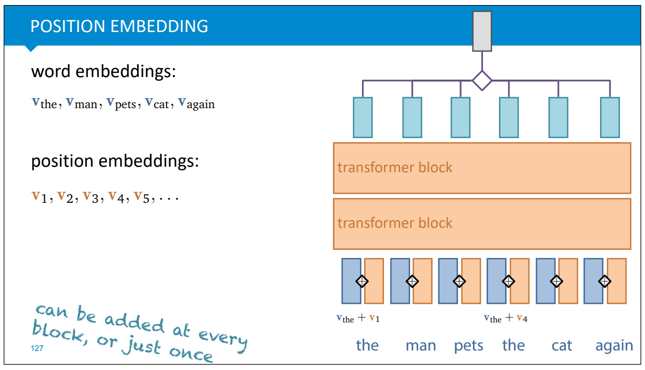
POSITION INFORMATION

This is not a real restaurant, it's a filthy burger joint.

This is not a filthy burger joint, it's a real restaurant.

To really interpret the meaning of the sentence, we need to be able to access the position of the words. Two sentences with their words shuffled can mean the exact opposite thing.

If we feed these sentences, tokenized by word, to the architecture on the right, their output label will necessarily be the same. The self-attention produces the same output vectors, with just the order differing in the same way they do for the two inputs, and the global pooling just sums all the vectors irrespective of position.



POSITION EMBEDDING

word embeddings:

$\mathbf{v}_{\text{the}}, \mathbf{v}_{\text{man}}, \mathbf{v}_{\text{pets}}, \mathbf{v}_{\text{cat}}, \mathbf{v}_{\text{again}}$

position embeddings:

$\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4, \mathbf{v}_5, \ldots$

can be added at every block, or just once

127

transformer block
transformer block
$\mathbf{v}_{\text{the}} + \mathbf{v}_1$
$\mathbf{v}_{\text{the}} + \mathbf{v}_4$
the man pets the cat again

The idea behind position embeddings is simple. Just like we assign each word in our vocabulary an embedding vector, we also assign each *position* in our vocabulary an embedding vector. This way, the input vectors for the first "the" in the input sequence and the second "the" are different, because the first is added to the position embedding $\mathbf{v}_1$ and the second is added to the input embedding $\mathbf{v}_2$.

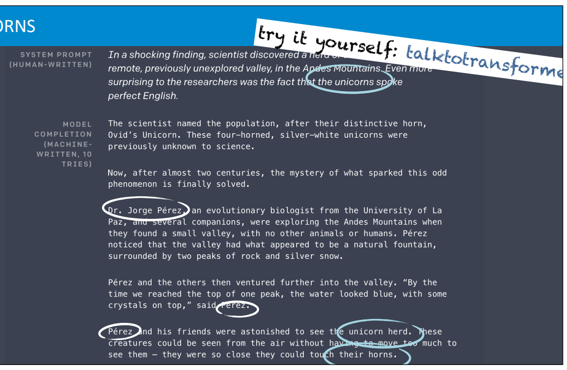
This break our equivariance: the position information becomes *part* of our embedding vectors, and is fed into the self attention. This is very effective, and very easy to implement. The only drawback is that we can't run the model very well on sequences that are longer than the largest position embedding observed during training.

RECAP

From self-attention to transformers:

- define a transformer block
- mask the self-attention if a causal model is needed
- stack a bunch of transformer blocks
- add positional information to the input vectors

UNICORNS



SYSTEM PROMPT (HUMAN-WRITTEN)

In a shocking finding, scientist discovered a herd of [illegible] remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.

MODEL COMPLETION (MACHINE-WRITTEN, 10 TRIES)

The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science.

Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved.

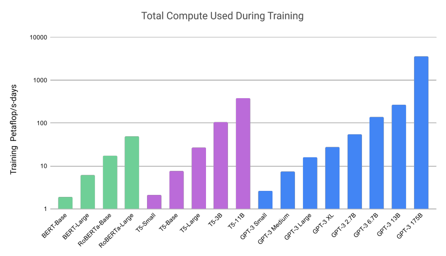
Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. Pérez noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow.

Pérez and the others then ventured further into the valley. "By the time we reached the top of one peak, the water looked blue, with some crystals on top," said Pérez.

Pérez and his friends were astonished to see the unicorn herd. These creatures could be seen from the air without having to move too much to see them – they were so close they could touch their horns.

---

total compute



Language models are few-shot learners. Brown et al 2019

Note the logarithmic scale.